SCALABLE AND ENERGY-EFFICIENT SIMT SYSTEMS FOR DEEP LEARNING AND DATA CENTER MICROSERVICES

by

Mahmoud Khairy A Abdallah

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

School of Electrical and Computer Engineering

West Lafayette, Indiana

August 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Timothy G. Rogers, Chair

Department of Electrical and Computer Engineering

Dr. Mithuna S Thottethodi

Department of Electrical and Computer Engineering

Dr. Milind Kulkarni

Department of Electrical and Computer Engineering

Dr. David Nellans

NVIDIA

Approved by:

Dr. Dimitrios Peroulis

ACKNOWLEDGMENTS

I have many people to thank who have supported me during my Ph.D. journey over the last five years. First and foremost, I would like to express my deepest and sincere gratitude to my advisor, Professor Tim Rogers. I really appreciate his expert guidance, meaningful mentorship, constant care, and support throughout my Ph.D.

I thank my committee members, Professor Mithuna S. Thottethodi, Professor Milind Kulkarni, and Dr. David Nellans, for supervising this thesis. Your feedback is integral to improving both the thesis and its constituent works. In particular, I would like to thank David for providing me with incredible technical support, valuable discussions, and countless pieces of critical advice during the LADM project.

I would like to thank my colleagues at AALP group who directly helped me with my research projects and social life, including Mengchi Zhang, Tsung Tai, Akshay Jain, Roland Green, Aaron Barnes, Ahmad Alawneh, Yechen Liu, Abhishek Bhaumick, Cesar Avalos, Junrui Pan, Fanjia Shen, Vadim Nikiforov, Jason Shen, Christin Bose, Ni Kang, and Weili An.

And last but not least, I would like to express my profound gratitude to my family for their unconditional and continued love and support. I am most grateful to my parents and brothers for enabling me to pursue my passion for learning and for being by my side at each step of this journey. I would also like to thank my friends in the Egyptian community at Purdue: Mohamed Zahran, Ashraf Youssef, Mostafa Abdallah, Tarek Ameen, Amr Ebid and my roommates Nour Hendy and Mohamed Saad.

Finally, I would like to acknowledge the funding sources that made my research possible: National Science Foundation, Purdue Graduate Fellowship, and Sandia National Labs.

PREFACE

The following is a list of my publications that have been incorporated into this dissertation in chronological order:

1. **Mahmoud Khairy**, Jason Shen, Tor M. Aamodt, and Timothy G. Rogers, "Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling," In The 47th International Symposium on Computer Architecture (**ISCA 2020**), Virtual Event, May 2020, (Acceptance rate: 77/421 = 18%)

2. **Mahmoud Khairy**, Vadim Nikiforov, David Nellans, and Timothy G. Rogers, "Locality-Centric Data and Threadblock Management for Massive GPUs," In The 53rd IEEE/ACM International Symposium on Microarchitecture (**MICRO 2020**), Virtual Event, October 2020 (Acceptance rate: 82/422 = 18%)

3. **Mahmoud Khairy**, Ahmad Alawneh, Aaron Barnes, and Timothy G. Rogers, "SIMR: Single Instruction Multiple Request Processing for Energy-Efficient Data Center Microservices", In The 55th IEEE/ACM International Symposium on Microarchitecture (**MICRO 2022**), Chicago, October 2022 (Acceptance rate: 86/348 = 22%)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

GPU	Graphics Processing Unit
GPGPU	General Purpose on Graphics Processing Unit
LADM	Locality Aware Data Management
SIMT	Single Instruction, Multiple Thread
SIMR	Single Instruction, Multiple Request
MIMD	Multiple-Instruction, Multiple-Data
SIMD	Single-Instruction, Multiple-Data
SMT	Simultaneous MultiThreading
RPU	Request Processing Unit
CPU	Central Processing Unit
HPC	High Performance Computing
ML	Machine Learning
DL	Deep Learning
AI	Artificial Intelligence
IoT	Internat of Things
NUMA	Non Uniform Memory Access
SM	Streaming Multiprocessor
L1	Level One
L2	Level Two
L3	Level Three
L1D	Level One Data
MSHR	Miss Status Holding Register
GDDR	Graphic Dual Data Rate
DRAM	Dynamic Random Access Memory
HBM	High Bandwidth Memory
BW	Bandwidth
CSR	Compressed Sparse Row
PDOM	Post Dominator

API	Application Programming Interface
TDP	Thermal Design Point
FLOPS	Floating point Operations Per Second
GB/sec	Giga Byte Per Second
QoS	Quality of Service

ABSTRACT

Moore's law is dead. The physical and economic principles that enabled an exponential rise in transistors per chip have reached their breaking point. As a result, High-Performance Computing (HPC) domain and cloud data centers are encountering significant energy, cost, and environmental hurdles that have led them to embrace custom hardware/software solutions. Single Instruction Multiple Thread (SIMT) accelerators, like Graphics Processing Units (GPUs), are compelling solutions to achieve considerable *energy efficiency* while still preserving *programmability* in the twilight of Moore's Law.

In the HPC and deep learning (DL) domain, the death of single-chip GPU performance scaling will usher in a renaissance in multi-chip Non-Uniform Memory Access (NUMA) scaling. Advances in silicon interposers and other inter-chip signaling technology will enable single-package systems, composed of multiple chiplets that continue to scale even as per-chip transistors do not. Given this evolving, massively parallel NUMA landscape, the placement of data on each chiplet, or discrete GPU card, and the scheduling of the threads that use that data is a critical factor in system performance and power consumption.

Aside from the supercomputer space, general-purpose compute units are still the main driver of data center's total cost of ownership (TCO). CPUs consume 60% of the total data center power budget, half of which comes from the CPU pipeline's frontend. Coupled with the hardware efficiency crisis is an increased desire for programmer productivity, flexible scalability, and nimble software updates that have led to the rise of software microservices. Consequently, single servers are now packed with many threads executing the same, relatively small task on different data.

In this dissertation, I discuss these new paradigm shifts, addressing the following concerns: (1) how do we overcome the non-uniform memory access overhead for next-generation multi-chiplet GPUs in the era of DL-driven workloads?; (2) how can we improve the energy efficiency of data center's CPUs in the light of microservices evolution and request similarity?; and (3) how to study such rapidly-evolving systems with an accurate and extensible SIMT performance modeling?

To this end, I propose three different frameworks and systems to address these challenges. First, to improve the quality of GPU research produced by the academic community, I have developed, *Accel-Sim*, a new GPU simulation framework to help solve the problem of keeping simulators up-to-date with contemporary designs. Using a counter-by-counter validation of the GPU memory system, Accel-Sim decreases cycle error from 94% in state-of-the-art simulation to 15%.

Second, to maintain GPU performance scalability in the twilight of Moore's Law, I propose a programmer-transparent Locality-Aware Data Management (LADM) system designed to operate on massive logical GPUs composed of multiple discrete devices, which are themselves composed of chiplets. LADM has two key components: a threadblock-centric compiler-assisted index analysis, and runtime system that performs adaptive data placement, threadblock scheduling and cache insertion policy. Compared to state-of-the-art multi-GPU scheduling, LADM reduces inter-chip memory traffic by 4× and capturing 82% of the unbuildable monolithic chip performance.

Third, to exploit the similarity in contemporary microservices, I propose a new class of computing hardware, the Request Processing Unit (RPU), which modifies out-of-order CPU cores to execute microservices using a Single Instruction Multiple Request (SIMR) execution model. Our solution leverages the CPU's programmability and latency optimizations while still exploiting the GPUs' SIMT efficiency and memory model scalability. By using the lock-step execution of concurrent requests, RPU reduces the size and access frequency to the out-of-order frontend and memory system. Our resulting RPU system processes 5.6× more Requests/Joule than multi-core CPUs while maintaining acceptable service latency and keeping the traditional microservices software stack unchanged.

1. INTRODUCTION

Single Instruction Multiple Thread (SIMT) hardware, like Graphics Processing Units (GPUs), has been widely adopted in many areas, including graphics, High-Performance Computing (HPC), and Deep Learning (DL). The Single Program Multiple Data (SPMD) pattern available in these workloads makes them amenable to lock-step execution on SIMT hardware, as threads execute the same program code and exhibit similar control flow. Moreover, these workloads show regular memory behavior, which increases memory coalescing opportunities. These program characteristics have led to significant performance and energy efficiency gains when they are ported to SIMT-based hardware, like GPUs. Thus, there is no wonder that a large portion of supercomputers found in the Top500 list relies on GPUs [1], [2]. *However, GPU performance scalability is at risk!* Building a larger GPU with dozens of GPU cores in one monolithic chip may not be possible due to low manufacture yield and high cost of building huge chips at small technology nodes [3].

Aside from supercomputer and HPC space, modern data centers exhibit massive degrees of *similar* request-level parallelism in which they receive a significant amount of independent requests from millions of users running the same service code. Further, the shift toward microservice and nanoservice-based architecture makes the control flow among these requests less divergent and reduces the cache footprint per node. Current CPUs run these requests independently on multi-core and do not exploit this similarity and finer granularity of microservices. These microservice requests follow the Single Program Multiple Data pattern with rich inter-request data sharing that can be efficiently leveraged on SIMT hardware. However, current SIMT hardware, like GPUs, is unsuitable for microservice as they optimize throughput over latency, making them ill-suited to meet the QoS requirement, and force the programmer to rewrite the microservices in GPGPU programming language (e.g., CUDA/OpenCL), hindering software productivity. Therefore, I argue for a new class of SIMT hardware that can take advantage of the latency-optimizations and programmability of the CPUs while still exploiting the SIMT efficiency and memory model scalability of the GPUs. In other words: *"Let's bring the SIMT Efficiency to the CPU world!"*.

1.1 Challenges

In this dissertation, I aim to address the following three challenges.

1.1.1 Inaccurate Out-of-date Academic Simulators

In computer architecture, significant innovation frequently comes from industry. However, the simulation tools used by industry are often not released for open use, and even when they are, the exact details of industrial designs are not disclosed. As a result, research in the architecture space must ensure that assumptions about contemporary processor design remain true. Figure 1.1 illustrates, simulation inaccuracy can lead to the retention of design proposals with overestimated benefits, or the rejection of design proposals with underestimated benefits. In one scenario promising ideas are throw out prematurely leading to less optimal solutions. In the other scenario, ineffective ideas are retained longer than necessary, leading to wasted time and effort during the architecture design process. Therefore, research cannot look ahead, if its baseline assumptions are too far behind.

To keep up with industry, state-of-the-art academic and public research must both be aware of and adapt to changes in contemporary designs. In the computer architecture space, keeping up with proprietary industrial machines is a challenge. This is a problem in all segments of the processor industry, but perhaps a more acute challenge in the programmable SIMT accelerator space, where the rapid scaling of parallelism, introduction of new processing pipelines (i.e. Tensor Cores [4]) and undocumented changes to both the microarchitecture and *Instruction Set Architecture* (ISA) are commonplace in each new product generation. GPU architectures have widely embraced the use of a virtual ISA (vISA), which provides hardware vendors with tremendous flexibility to make machine ISA (mISA) changes while still maintaining binary compatibility. Vendors like NVIDIA keep the operation of these ISAs private, while others like AMD document each new machine ISA, but freely make drastic changes which open-source simulators must then implement.

This situation presents three separate, but related challenges: (1) How do academic researchers quickly simulate a new, often undocumented ISA every year and a half? (2) Once functionally correct, how are changes to the architecture detected and modeled? (3)

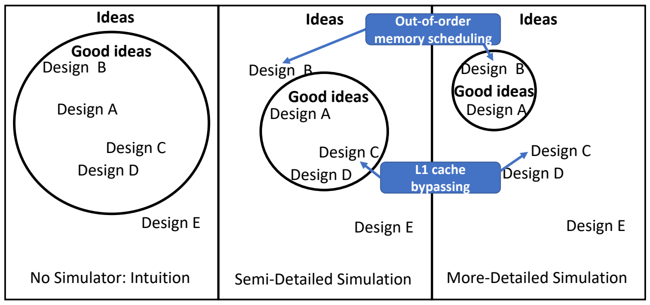


Figure 1.1. The architecture design and simulation process represented as a collection of good and ineffective ideas. As the level of simulation detail increases, the space of effective ideas shrinks and potentially moves.

What is a sustainable, rigorous validation mechanism to ensure that new baselines are still tracking industrial designs?

1.1.2 Post-Moore Multi-GPU Multi-Chiplet Scaling

GPU accelerated workloads are commonly used in deep learning and exascale HPC computing systems [5], [6]. These workloads exhibit high levels of implicit parallelism, which transparently enables performance scalability, but only if GPUs can continue to scale their hardware resources into the future. Over the past decade GPUs have more than quadrupled the number of Streaming Multiprocessors (SMs) in their designs, while simultaneously increasing their on-chip transistors by an order of magnitude. Figure 1.2 shows the anticipated performance of balanced GPU resource scaling (Streaming Multiprocessor (SM), SM-interconnect bandwidth, registers, caches, and DRAM bandwidth) if all components can be equally scaled into the future. While the expected performance is appealing, building a GPU with hundreds of SMs in a single monolithic GPU will not be possible due to low manufacturing yields and the high cost of building large chips at small technology nodes [3], [7].

To overcome these problems and enable continuous performance scaling beyond the bounds of Moore's law [8], [9], researchers have proposed increasing GPU transistor count by both aggregating multiple GPUs together (coordinated as a single virtual GPU), as well as dissagregating single-GPUs into scalable multi-chip-module designs [3], [10]. In both ap-

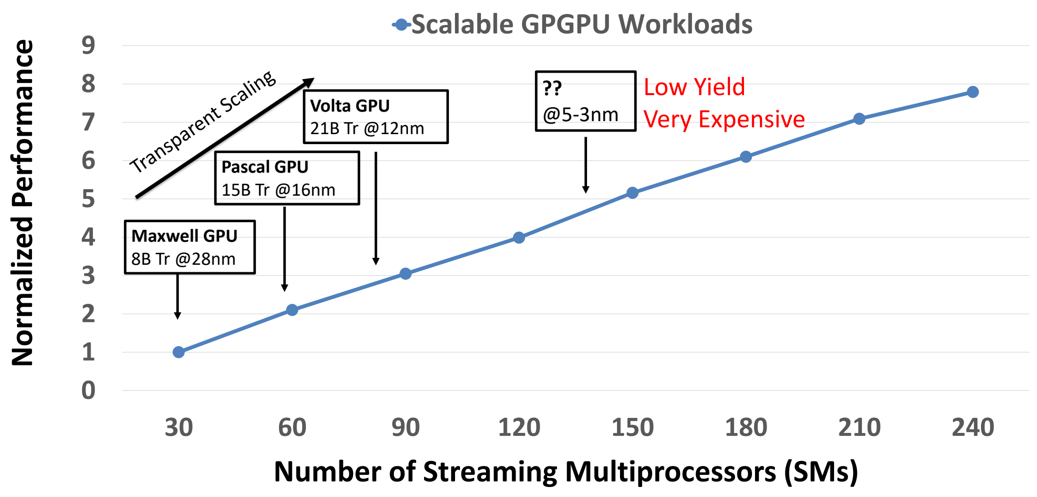


Figure 1.2. Performance of GPU workloads in a hypothetical monolithic GPU, where all resources scale proportionally.

proaches, to maintain the existing single GPU programming model and support transparent scaling for current CUDA programs, the architecture and runtime systems must coordinate to hide the fact that a single programmer visible GPU may be comprised of several different GPU NUMA domains. Maintaining this illusion enables rapid software development on small local GPU resources while enabling scalable performance on larger and more complex GPU interconnect topologies. Consequently, transparently overcoming NUMA effects will be one of the largest problems facing GPUs in both these integration domains over the next decade.

1.1.3 Data Center Energy Efficiency Crisis and Microservices Evolution

The growth of hyperscale data centers has steadily increased in the last decade, and is expected to continue in the coming era of Artificial Intelligence and the Internet of Things [14]. However, the slowing of Moore's Law [9] has resulted in energy [15], environmental [16], [17] and supply chain [18] issues. It is anticipated that, by 2030, the data centers will consume about 10% of the total electricity demand [19]. These all have lead data centers to embrace custom hardware/software solutions [20], [21].

While improving Deep Learning (DL) inference has received significant attention [20], [22], general purpose compute units are still the main driver of a data center's total cost

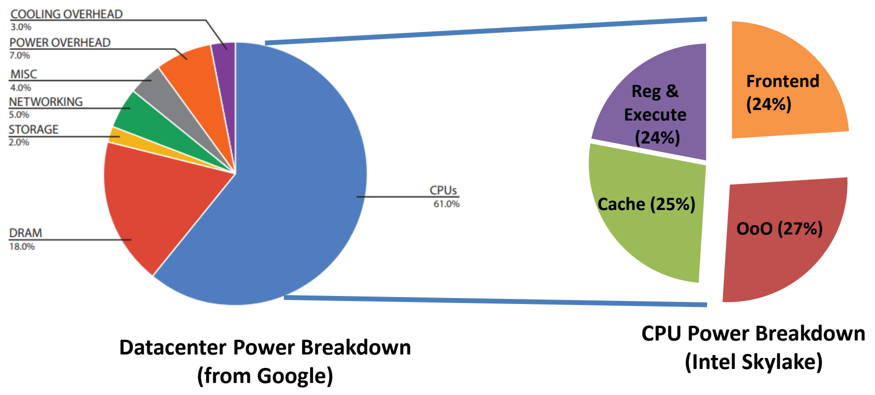


Figure 1.3. Data center power consumption breakdown. Source: [11]–[13]

of ownership (TCO). Figure 1.3 demonstrates that CPUs consume 60% of the data center power budget [11], half of which comes from the pipeline's frontend (i.e. fetch, decode and Out-of-Order (OoO) structures) [12], [13], [23]–[25]. Therefore; 30% of the data-center's total energy is spent on CPU instruction supply.

Coupled with the hardware efficiency crisis is an increased desire for programmer productivity, flexible scalability and nimble software updates that has lead to the rise of software microservices. Monolithic server software has been largely replaced with a collection of micro and nanoservices that interact via the network [26]–[28]. Compared to monolithic services, microservices spend much more time in network processing [27], [29], have a smaller instruction and data footprint [27], and can suffer from excessive context switching due to frequent network blocking [26], [30]–[32]. Microsecond scale network latencies cannot be hidden by course-grained OS context switching [33], requiring more threads on each core to support higher throughput [34], [35], under tight single-thread latency constraints.

1.2 My Approach

In this subsection, I discuss my proposed approaches to overcome and solve the previous hurdles.

1.2.1 Building for Flexibility and Modularity: An Accurate and Extensible Simulation Framework

To help bridge the gap between opaque industrial innovation and public research, I introduce three mechanisms that make it much easier for GPU simulators to keep up with industry. First, I introduce a new GPU simulator frontend that minimizes the effort required to simulate different machine ISAs through trace-driven simulation of NVIDIA's native machine ISA, while still supporting execution-driven simulation of the virtual ISA. Second, I extensively modify GPGPU-Sim's performance model to increase the level of detail, configurability and make hardware validation easier. Along with the detailed statistics available in simulation, the updated model outputs a set of performance counters that have 1:1 analogs with profiling data generated by contemporary GPU profilers. Finally, surrounding the new simulator and flexible performance model is an infrastructure that enables quick, detailed validation. A comprehensive set of microbenchmarks and automated correlation plotting make performance model validation an automated process.

I use these three new mechanisms to build *Accel-Sim*, an accurate, detailed simulator capable of modeling the performance of contemporary GPUs with 80.2% less error than state-of-the-art, open-source simulators over a wide range of 78 workloads, consisting of 2221 kernel instances. I further demonstrate that Accel-Sim is able to simulate benchmark suites that no other open-source simulator can. In particular, I demonstrate Accel-sim's ability to execute 27 workloads, comprised of 2328 kernel instances, from the machine learning benchmark suite Deepbench. Deepbench makes use of closed-source, hand-tuned kernels with no virtual ISA implementation. To demonstrate Accel-Sim's flexibility and extensibility, I model and validate Accel-Sim against four different NVIDIA GPU generations. Accel-Sim is the only open-source simulator to support contemporary SASS simulation.

Finally, to highlight the effects of falling behind industry, this thesis presents two case-studies that demonstrate how incorrect baseline assumptions can hide new areas of opportunity and lead to potentially incorrect design decisions.

1.2.2 Optimizing for Locality: Transparent Multi-GPU Scaling for DL and HPC Workloads

In order to maintain GPU performance scalability for deep learning and HPC workloads, I propose a programmer-transparent *Locality-Aware Data Management* (LADM) system designed to operate on massive logical GPUs composed of multiple discrete devices, which are themselves composed of chiplets. LADM has three key components: a threadblock-centric index analysis, a runtime system that performs data placement and threadblock scheduling, and an adaptive cache insertion policy. The runtime combines information from the static analysis with topology information to proactively optimize data placement, threadblock scheduling, and remote data caching, minimizing off-chip traffic. Compared to state-of-the-art multi-GPU scheduling, LADM reduces inter-chip memory traffic by 4× and improves system performance by 1.8× on a future multi-GPU system.

1.2.3 Exploiting Microservices Similarity and Eliminate Redundancy: Single Instruction Multiple Request Processing

Contemporary data center servers process thousands of similar, independent requests per minute. In the interest of programmer productivity and ease of scaling, workloads in data centers have shifted from single monolithic processes on each node toward a micro and nanoservice software architecture. As a result, single servers are now packed with many threads executing the same, relatively small task on different data.

State-of-the-art data centers run these microservices on multi-core CPUs. However, the flexibility offered by traditional CPUs comes at an energy-efficiency cost. The Multiple Instruction Multiple Data execution model misses opportunities to aggregate the similarity in contemporary microservices. I observe that the Single Instruction Multiple Thread execution model, employed by GPUs, provides better thread scaling and has the potential to reduce frontend and memory system energy consumption. However, contemporary GPUs are ill-suited for the latency-sensitive microservice space.

To exploit the similarity in contemporary microservices, while maintaining acceptable latency, I propose the Request Processing Unit (RPU). The RPU combines elements of traditional out-of-order CPUs with lockstep thread aggregation mechanisms found in GPUs to execute microservices in a Single Instruction Multiple Request (SIMR) fashion. To complement the RPU, I also propose a SIMR-aware software stack that uses novel mechanisms to batch requests based on their predicted control-flow, split batches based on predicted latency divergence and map per-request memory allocations to maximize coalescing opportunities. My resulting RPU system processes 5.6× more Requests/Joule than multi-core CPUs, while maintaining acceptable service latency

1.3 Thesis Statement

My thesis statement, hence, is as follows:
SIMT-based accelerators, like GPUs and my proposed RPUs, are promising solutions to achieve significant energy efficiency while still preserving programmability in the twilight of Moore's Law. I propose three approaches to build next-generation scalable and energy-efficient SIMT systems: (1) detect and optimize for each type of locality exist in the DL and HPC workloads to overcome NUMA effects and reduce off-chip communication between multi-chiplet GPUs, (2) exploit microservices execution similarity and eliminate redundancy to improve data center energy efficiency, and (3) build extensible and validated SIMT simulation tools to keep-up with industrial changes.

1.4 Contributions

This dissertation makes the following contributions:

- It introduces Accel-Sim, a simulation framework explicitly designed to make modeling and validating future GPUs easier. By utilizing a flexible frontend, capable of switching between execution-driven vISA simulation and trace-driven mISA simulation, I am able to simulate hand-tuned machine code from NVIDIA binaries without giving up the option to perform execution-driven simulation when appropriate. I demonstrate Accel-Sim's flexibility by modeling GPUs from Kepler to Turing.

- It performs a rigorous, counter-by-counter comparison of my new performance model against state-of-the-art simulation when modeling an NVIDIA Volta card, reducing error by 79 percentage points. Through this analysis, we uncover and model several previously undocumented architectural features in contemporary GPUs.

- It introduces a comprehensive set of validation infrastructure that includes: a set of targeted GPU microbenchmarks, an automated parameter tuner and an automatic correlation visualizer, useful for adapting Accel-Sim to model new designs quickly.

- It performs a set of case-studies to concretely demonstrate the effects of falling behind industry and uses Accel-Sim to suggest future areas of new research for GPUs.

- It performs a detailed analysis of the locality types present in GPU programs and show that no state-of-the-art NUMA-GPU system can exploit them all. I propose LADM, which uses static index analysis to inform runtime data placement, threadblock scheduling, and remote caching decisions by exploiting a new logical abstraction called the *GPU datablock*.

- It leverages this automatic analysis to perform threadblock and datablock co-placement within hierarchical GPUs. By pre-calculating an optimized data layout in the compiler, LADM can orchestrate prefetching that negates on-demand page-faulting effects and adjust the threadblock schedule based on dynamic data structure sizes.

- Building on my program analysis, I architect a novel compiler-informed cache organization that selectively inserts requests into each L2 cache partition based on the memory request's origin relative to the data's home node and its likelihood for reuse. By understanding the expected reuse patterns for datablocks, LADM's cache hierarchy minimizes both inter-GPU and inter-chiplet bandwidth, the primary factor influencing the scalability of future GPUs.

- It performs the first SIMT-efficiency characterization of microservices using their native CPU binaries. I demonstrate that, given the right batching mechanisms, microservices execute efficiently on SIMT hardware.

- It proposes a new hardware architecture, the Request Processing Unit (RPU). The RPU improves the energy-efficiency and thread-density of contemporary OoO CPU cores by exploiting the similarity between concurrent microservice requests. With a high SIMT efficiency, the RPU captures the single-threaded advantages of OoO CPUs, while increasing Requests/Joule.

- It proposes a novel software stack, co-designed with the RPU hardware that introduces SIMR-aware mechanisms to compose/split batches, tune SIMT width, and allocate memory to maximize coalescing.

- On a diverse set of 13 CPU microservices, I demonstrate that the RPU improves Requests/Joule by an average of 5.6x versus OoO single threaded and SMT CPU cores, while maintaining acceptable end-to-end latency.

1.5 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 introduces Accel-Sim, a new GPU simulation framework that helps solve the problem of keeping simulators up-to-date with contemporary designs. I show how the proposed simularot is extensible, accurate and validated to various GPU generation found today in the market.

- Chapter 3 details, Locality-Aware Data Management (LADM), a transparent and scalable system designed to operate on massive logical GPUs composed of multiple discrete devices, which are themselves composed of chiplets.

- Chapter 4 performs SIMT-efficiency characterization of microservices and proposes Single Instruction Multiple Request system, a hardware-software co-design to exploit request similarity in the data centers.

- Chapter 5 discusses related work.

- Chapter 6 concludes the dissertation and discusses directions for potential future work.

2. ACCEL-SIM: AN EXTENSIBLE SIMULATION FRAMEWORK FOR VALIDATED GPU MODELING

In computer architecture, significant innovation frequently comes from industry. However, the simulation tools used by industry are often not released for open use, and even when they are, the exact details of industrial designs are not disclosed. As a result, research in the architecture space must ensure that assumptions about contemporary processor design remain true. We present a new simulation framework, Accel-Sim, that is designed to address this challenge.

Accel-Sim introduces a flexible frontend, that enables it to operate in either trace- or execution-driven mode. Accel-Sim includes a trace-generation tool (built using the NVBit [36] binary instrumentation tool), that produces machine ISA instruction traces from any CUDA binary, including those that use closed-source libraries, like cuDNN [37]. These machine ISA traces are then converted into an ISA-independent intermediate representation that is input to the performance model. The trace-driven frontend allows Accel-Sim to simulate the machine ISA in new cards without implementing the ISA's functional model and increases the accuracy of the simulator over executing the virtual ISA. However, trace-based simulation has its drawbacks. Evaluating new designs that rely on the data values stored in registers or memory [38] and global synchronization mechanisms [39] are either not possible or very difficult to study without emulation-based execution-driven simulation. Since Accel-Sim is based on an ISA-independent performance model, it is also capable of running emulation-based execution-driven simulations using NVIDIA's relatively stable, well documented virtual ISA, PTX. Accel-Sim is the first academic simulation framework to support contemporary CUDA applications, modern *Source and ASSembly* (SASS) machine ISAs and simulate hand-tuned assembly in closed-source GPU binaries.

Shifting the focus from implementing an undocumented functional model enables Accel-Sim to focus on the performance model. We extensively modify GPGPU-Sim's performance model (which we release as part of GPGPU-Sim 4.0) to increase its level of detail, configurability and accuracy. To facilitate rigorous validation, the performance model outputs a set of counters that have 1:1 equivalents with hardware data emitted by NVIDIA profilers, in addition to the detailed statistics that can only be provided by simulation. These counter values are then fed into an automated tuning framework that generates correlation plots and modifies the simulator's configuration files, making it easier to create a validated performance model for a new GPU.

Using our new frontend and validation infrastructure, we perform an extensive modeling effort that builds on GPGPU-Sim's [40] performance model to create a more flexible, extensible GPU simulator. Through careful counter-by-counter validation, we expose several changes to contemporary GPU hardware. We demonstrate Accel-Sim's flexibility and accuracy by validating it against four generations of NVIDIA GPUs ranging from Kepler to Turing, performing an extensive side-by-side comparison of Accel-Sim and GPGPU-Sim 3.x modeling a Volta V100 [41].

During the course of this analysis we uncover a number of interesting insights into how contemporary hardware works. We utilized every publicly available resource to construct the performance model, capturing the details of what others have either disclosed or discovered [42]–[45]. In the process of correlating the memory system we discover and model several undocumented features, such as: details of the streaming, adaptive L1 data cache, sectoring of both the L1 and L2, a sub-warp, sector-centric memory coalescing policy, and an L2 write policy that conserves memory bandwidth in the presence of sub-sector reads.

Finally, this work performs two case-studies to highlight new research opportunities on contemporary GPUs and to demonstrate pitfalls that are alleviated by having a simulation infrastructure that is easier to validate. In particular, we demonstrate that out-of-order memory access scheduling, which appears relatively ineffective using an older model, yields a significant performance improvement when the level of detail in the memory system is increased to more closely match hardware. We also demonstrate that the L1 cache throughput bottleneck present in GPUs no longer exists in contemporary hardware, decreasing the effectiveness of techniques that selectively bypass the L1 for throughput reasons.

Table 2.1. Landscape of open-source GPU simulation. Accuracy numbers are taken from each simulator's respective publication.

	GPGPU-Sim 3.x [40]	Gem5-APU [48], [49]	MGPU-Sim [50]	MacSim [51], [52]	Multi2-Sim [53], [54]	**Accel-Sim**
ISA	vISA + mISA GT200	vISA + AMD mISA	vISA + AMD mISA	vISA + Intel mISA	vISA + mISA Kepler + mISA AMD	vISA + NVBit generated [36] mISA
Front-end	Execution	Execution	Execution	Trace	Execution	Trace- and Execution-driven
Validated perf. model	Fermi	AMD	AMD	Fermi	Kepler	Kepler, Pascal, Volta, Turing
Validated Workloads	14	10	7	N/A	24	80
Reported Accuracy (error %)	35% [46]	42% [49]	5.5% [50]	N/A	19% [53]	15%
Simulation rate (KIPS)	3	N/A	28 [50]	N/A	0.8 [50]	12.5 (Trace-driven) 6 (Exec-driven)
Multi-threaded simulation	✗	✗	✓	✗	✗	✗
Hand-tuned NVIDIA libraries (i.e. Volta cuDNN)	✗	✗	✗	✗	✗	✓

2.1 Background

In contemporary computer architecture research, simulation is commonly used to estimate the effectiveness of a new architectural design idea. High-level simulators enable architects to rapidly evaluate ideas at the expense of less accurate simulation results. Ideas that do not show promise in simulation are discarded while those that do show promise are refined in an iterative process.

Our work focuses on the simulation of massively parallel architectures, in particular GPUs. GPUs have witnessed rapid change and a widespread increase in their adoption with the rise of GPGPU computing and machine learning. In academia, the design of programmable accelerators is mostly carried out through modeling new techniques in high level GPU simulators. Over the past four years, there have been approximately 20 papers per year focusing on GPU-design at the top architecture conferences. 80% of those papers have used today's most popular open-source GPU simulator, GPGPU-Sim [46]. The relative popularity of GPGPU-Sim can be attributed to several factors, but it's most appealing aspect is perhaps the accuracy with which it models modern GPUs (relative to other open-source solutions). Such accuracy should provide a solid baseline for studying important architectural ideas that are relevant to future machines. Recent work on validating GPGPU-Sim [47] has demonstrated that there are several areas where a lack of detail in the performance model creates a a major source of error. However, the bulk of the error comes from the modeling of the memory system.

Table 2.1 presents a survey of the open-source GPU simulation space. Over the last decade, several GPU simulators have been developed, each serving a different purpose. MacSim [51] is an early GPU simulator that performs trace-based simulation of NVIDIA's virtual ISA and includes a Fermi-like microarchitecture model. Recent work on MacSim has extended the simulator to execute traces for Intel GPUs [52]. Multi2-Sim is a versatile GPU simulator that has emulation-based functional simulation for both the virtual ISA and a subset of older machine ISAs from both AMD and NVIDIA. GPGPU-Sim [40] is a CUDA-centric simulator capable of functionally executing NVIDIA's virtual ISA and a subset of an older machine ISA. More recently, Gem5 has been augmented to support an APU performance model for AMD's virtual and machine ISAs [48]. MGPU-Sim [50] is a parallel GPU simulator for AMD's virtual and machine ISAs. One of the primary drawbacks with all the simulators that support execution-driven machine ISAs is the challenge of keeping up with changes to the mISA's functional model. As a result, only a limited subset of the instruction set (and hence applications) are ever fully supported. This is especially important for supporting optimized libraries that often use exotic, hand-tuned machine code to improve performance, such as NVIDIA's cuDNN. Although recent work has augmented GPGPU-Sim [55] to enable virtual ISA execution of these libraries, the functionality only works in pre-Volta GPUs. For Volta and Turing, cuDNN executes hand-tuned SASS kernels for which there is no virtual ISA implementation. Accel-Sim's support for machine ISA traces bypasses all of these issues. In addition, existing GPU simulation frameworks lack a systematic methodology to validate and model new architectural designs quickly.

2.2 Accel-Sim Simulation Framework

Figure 2.1 depicts an overview of our new simulation framework. Accel-Sim is composed of: (1) a flexible frontend that supports execution-driven simulation of NVIDIA's virtual ISA and trace-driven simulation of contemporary machine ISAs, (2) a flexible and detailed performance model, (3) a correlation generation tool, and (4) a microbenchmarks-based configuration tuner. Together, the four components reduce the effort required to model contemporary and future GPUs.

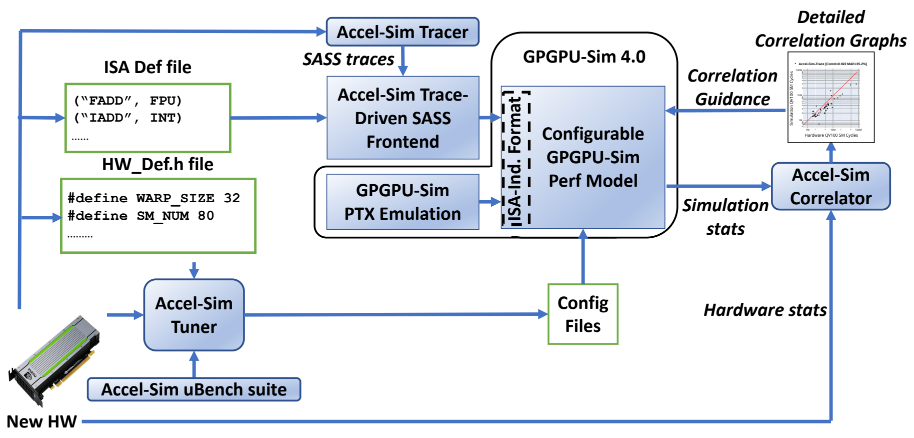


Figure 2.1. Accel-Sim's end-to-end flow.

Table 2.2. Example demonstrating how mISA instruction traces and vISA instructions translate into the ISA-independent intermediate representation used by the performance model. In traces, the mapping between opcode and execution unit is provided by the *ISA def* file in Figure 2.1.* *indicates these values are computed using emulation.*

Execution Mode	Example Instruction	ISA-independent representation					
		PC	Active mask	Reg info: (dsts, srcs)	Exec. unit	Memory addresses, width	Memory scope
[Trace] Kepler mISA	LD.E R4, [R6]	0x78	0xFFFF	R4, R6	Memory unit	0x2000,..., 4	global, L1 cached
[Trace] Volta mISA	LDG.E.U32.SYS R4, [R6]	0x32	0x00FF	R4, R6	Memory unit	0x4000,..., 4	global, L1 cached
[Trace] Pascal mISA	IADD.X R7,R7,R0	0x12	0x00EF	R7, R0	INT unit	-	-
[Exec] PTX vISA	ld.global.cg.b32 r1, [r0];	*	*	R1, R0	Memory unit	*	global, L2 cached

2.2.1 Flexible Frontend

Our new frontend supports both vISA (PTX) execution-driven and mISA (SASS) trace-driven simulation. In trace-driven mode, mISA traces are converted into an ISA-independent intermediate representation, that has a 1:1 correspondence to the original SASS instructions. Table 2.2 depicts an example of how SASS instructions from different machine generations and the virtual instructions from PTX are translated into the ISA-independent representation used by the performance model. The intermediate format is integrated into GPGPU-Sim 4.0 and represents the interface between the frontend and the performance model. The format includes all the information necessary to perform timing simulation, in particular: (1) the instruction's control flow (PC and active mask), (2) the instruction's datapath information (registers accessed and execution unit) and (3) memory addresses for ld/st instructions. In execution-driven mode, the active mask and memory addresses are computed by emulating the PTX instructions, whereas these values are embedded in the trace when executing the machine ISA.

We generate the traces from NVIDIA GPUs using Accel-Sim's tracer tool that is built on top of NVbit [36]. We use base+stride compression for the memory traces to keep the trace sizes within an acceptable range. When a new SASS ISA is released, users provide the frontend with an *ISA Def* file that specifies where each instruction should be executed. This is a relatively simple mapping that can be derived from publicly available information on NVIDIA's machine ISA [56].

By not emulating hundreds of scalar threads each cycle, Accel-Sim's trace-driven mode improves simulation speed versus execution-driven mode. Prior work [47], [49] demonstrates that executing GPU vISAs may not be an accurate representation of some workloads. The mISA representation of the program includes register allocation and other compiler optimizations, whereas the vISA assumes an infinite register space and has naive instruction scheduling. Further, supporting SASS gives researchers the ability to simulate closed-source, optimized libraries, such as cuDNN [37], which are written in hand-tuned SASS.

2.2.2 Flexible and Detailed Performance Model

To accurately model contemporary GPUs, we make extensive modifications to GPGPU-Sim 3.x's performance model. This new performance model is released as part of GPGPU-Sim 4.0 and can be used, independent of Accel-Sim, for PTX simulation. Accel-Sim utilizes our GPGPU-Sim 4.0 performance model, interfacing with it through the ISA-independent instruction representation.

Figure 2.2 depicts an overview of the performance model. Streaming multiprocessors (SMs) are composed of multiple warp schedulers. Each warp scheduler has a dedicated register file (RF) and its own execution units (EUs). A warp scheduler with its dedicated RF and EUs is called a sub-core [4], [57]. Sub-cores are isolated, sharing only the instruction cache and the memory subsystem. The memory coalescing unit is designed to support sub-warp coalescing on every consecutive group of N threads. Our performance model is capable of simulating both separate and unified L1D cache and shared memory scratchpad [58].

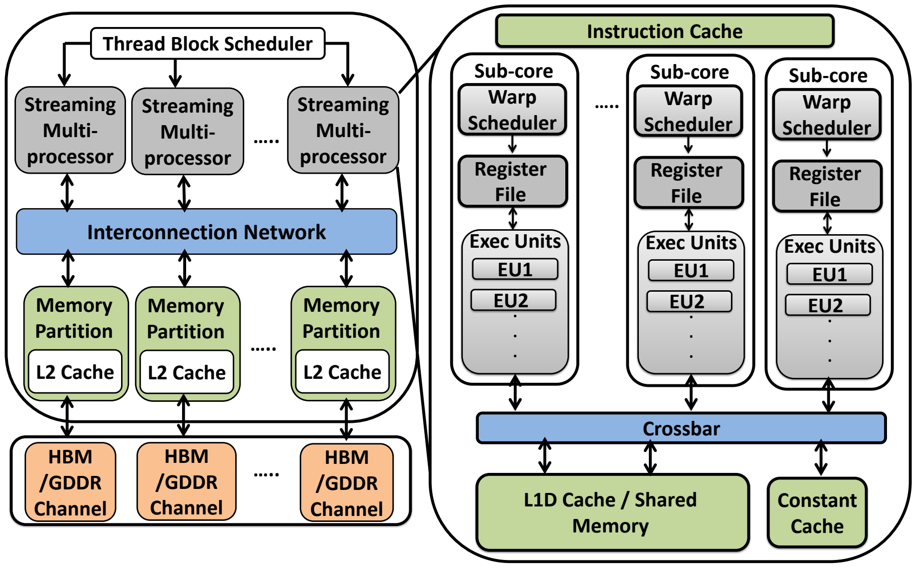
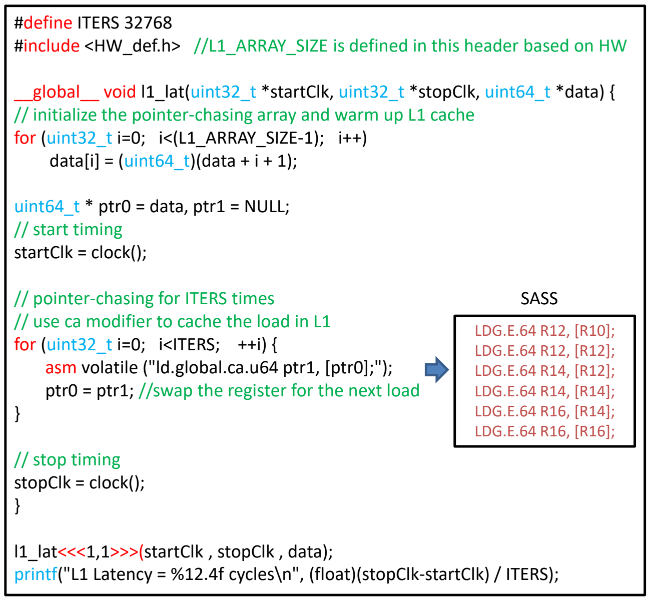


Figure 2.2. Updated GPGPU-Sim 4.0 performance model.

Contemporary GPU architectures make use of an *adaptive cache* mechanism, in which the device driver transparently configures the shared memory capacity and L1D capacity on a per-kernel basis. Using the adaptive cache, if a kernel does not utilize shared memory, all the on-chip storage will be assigned to the L1D cache [59].

With the slowing growth of Moore's law, domain-specific processing pipelines continue to be introduced in GPUs (i.e. Tensor cores in Volta [41]). To ensure extensibility with this trend, Accel-Sim supports adding simple execution units from the configuration file, without the need to update the codebase. To add a new specialized unit, the user declares the new unit in the configuration file and maps the machine ISA op codes that use this unit in the *ISA def* file, as described in Figure 2.1. If execution-driven PTX support is required, then the user must also implement the code to emulate the new instruction's functionality. To determine the latency and throughput of the new unit, the user specifies a sequence of instructions that can be measured by the microbenchmark tuning framework (described in Section 2.2.3). We follow this process to model Volta and Turing's Tensor Cores.

Our GPU cache model supports a throughput-oriented, banked, and sectored cache design [60], [61]. The cache is flexible enough to model GPUs from Kepler through Turing. We also model the CPU-GPU memory copy engine, since all DRAM accesses go through the L2, including CPU-GPU memory copies [62]. To reduce unbalanced memory accesses across



```
#define ITERS 32768
#include <HW_def.h>   //L1_ARRAY_SIZE is defined in this header based on HW

__global__ void l1_lat(uint32_t *startClk, uint32_t *stopClk, uint64_t *data) {
// initialize the pointer-chasing array and warm up L1 cache
for (uint32_t i=0;   i<(L1_ARRAY_SIZE-1);   i++)
      data[i] = (uint64_t)(data + i + 1);

uint64_t * ptr0 = data, ptr1 = NULL;
// start timing
startClk = clock();

// pointer-chasing for ITERS times
// use ca modifier to cache the load in L1
for (uint32_t i=0;   i<ITERS;    ++i) {
     asm volatile ("ld.global.ca.u64 ptr1, [ptr0];");
     ptr0 = ptr1; //swap the register for the next load
}

// stop timing
stopClk = clock();
}

l1_lat<<<1,1>>>(startClk , stopClk , data);
printf("L1 Latency = %12.4f cycles\n", (float)(stopClk-startClk) / ITERS);
```

```
SASS
LDG.E.64 R12, [R10];
LDG.E.64 R12, [R12];
LDG.E.64 R14, [R12];
LDG.E.64 R14, [R14];
LDG.E.64 R16, [R14];
LDG.E.64 R16, [R16];
```

Figure 2.3. L1 latency microbenchmark

L2 memory partitions, which we refer to as partition camping [63], [64], we add advanced partition indexing that xors the L2 bank bits with randomly selected bits from the page row bits using a Galois-based irreducible polynomial (IPOLY) interleaving mechanism [65]. Partition camping is a major problem in contemporary GPUs that have 2^n memory partitions, like High Bandwidth Memory (HBM) [66] which has 8 channels per stack [64], [67]. IPOLY hashing is guaranteed to be conflict-free for all 2^n strides, which are common in GPGPU applications, and also shows reasonable, deterministic performance for other strides [65].

In the memory system, we model new advances in HBM and GDDR6 [66], [67]. This includes the dual-bus interface to launch row and column commands simultaneously, increasing bank parallelism [67], and detailed HBM timing. Further, we implement well-known memory optimization techniques such as advanced xor-based bank indexing and separate read/write buffers, to reduce memory bank conflicts and read-write interference [68]–[70].

2.2.3 Tuner and Targeted Microbenchmarks

Accel-Sim includes a microbenchmark suite to pinpoint latency, bandwidth and geometry changes to known hardware structures from one generation to another. When new hardware is released, the user provides a *hardware def* header file to the tuner, as shown in Figure 2.1. The file is used by the microbenchamrks to help discover non-public configuration parameters. The def file enumerates a minimal amount of information (such as number of SMs, warp size, etc.) that can be derived from publicly available documents [41].

In total, we have 38 microbenchmarks that span from L1/L2/shared memory latency and attained bandwidth, cache write policy, cache configuration, access/sector granularity, number of cache banks, memory coalescing policy, cache streaming behavior, execution unit latency/throughput and DRAM latency/bandwidth for different data elements: float, double and 128-bit vector. To illustrate the general configuration of the microbenchmarks, Figure 2.3 lists the code for our L1 latency microbenchmark that uses pointer chasing [44], [71] to create data dependencies with minimal overhead. The microbenchmark is written such that the kernel's execution time is dominated by the L1 latency. After execution, the tuner reads the microbenchmark's output and generates a configuration file for the performance model.

For other parameters that cannot be directly determined by our microbenchmarks (such as warp scheduling, memory scheduling, the L2 cache interleaving granularity and the L2 cache hashing function), Accel-Sim simulates each possible combination of these four parameters on a set of memory bandwidth microbenchmarks (l1-bw, l2-bw, shd-bw and mem-bw). The combination with the highest average hardware correlation is chosen by the tuner.

2.2.4 Correlator

An automated configuration tuner does not capture more drastic architectural changes in different generations. Manual effort is required to model such changes. However, the magnitude of the effort can be lessened with quickly generated, targeted information on inaccuracy. Accel-Sim's correlation tool automates the process of generating counter-by-counter correlation data for different architectures. The tool generates graphs and data that serve as correlation guidance to pin-point workloads and metrics where the simulator is not well correlated to hardware. Using insights from the correlation of various performance counters over different realistic workloads and microbenchmarks, performance bugs or misrepresentations in the simulator are identified and corrected.

2.2.5 Simulation Rate

Adding detail and flexibility to a performance model often comes at the expense of increased simulation time. Although Accel-Sim is not multithreaded, we take steps to improve its speed. With our improvements, Accel-Sim in trace-driven mode is able to perform 12.5 kilo warp instructions per second, a 4.3× simulation time improvement over GPGPU-Sim 3.x. Half of our speed improvement comes from the fact that trace-driven mode avoids functional execution. The second half comes from a simulation optimization strategy we call *simulation-gating*, which is a trade-off between event-driven and cycle-driven simulation. GPU simulations involve thousands of in-flight memory requests and hundreds of active threads. Thus, there will always be something to simulate each cycle. Similar to GPGPU-Sim 3.x, the updated performance model is cycle-driven, ticking all the components every cycle. However, we observe that the simulator ticks many empty components and spends a significant amount of time doing non-useful work. To overcome this issue, we keep a status state for each component (core, execution unit, cache, dram channel), then the simulator only ticks active components every cycle and skips unnecessary code and loop iterations when there is no useful work in the pipeline. Finally, in trace-driven mode, the user can set kernel-based checkpointing to select the desired hotspot kernels to simulate and avoid executing long running initialization kernels, which can further improve simulation speed.

2.3 Workloads

To validate Accel-Sim and perform a detailed comparative analysis against prior work, we use a wide range of applications from different domains and benchmark suites. With the exception of a handful of applications with trace-size issues, we run all the applications found in Rodinia 3.1 [72], the CUDA SDK [73], Parboil [74], Polybench [75], CUTLASS [76], our microbenchmark suite, and cutting-edge machine learning workloads from the Deepbench benchmark suite [77]. The sum total is 140 workloads from the applications listed in Table 3.4. The applications *cfd*, *heartwall*, *hotspot3D*, *huffman*, *leukocyte*, *srad_v2* from Rodinia, *lbm* and *tpacf* from Parboil, *corr*, *cover*, *fdtd2d* and *gram-shm* from Polybench are omitted since their trace sizes are prohibitively large. We leave developing trace compression

Table 2.3. Workloads used in our study. Table 2.8 lists the hardware cycles for each workload.

Benchmark Suite	Workloads	Trace Size [Compressed]
Rodinia 3.1 [72]	b+tree, backprob, bfs, hotspot, srad-v1, sc, nn, needle, dwt2d, lud, lavaMD, kmeans, myocyte	302 GB [15 GB]
CUDA SDK [73]	sp, blk, vec-add, traspose, conv, scan, sorting-net, walsh-transform, histo, mergesort	18 GB [1.2 GB]
Parboil [74]	sad, sgemm, stencil, cutcp, mri-q, histo, spmv, bfs	250 GB [12 GB]
Polybench [75]	2dconv, 3dconv, 2mm, 3mm, mvt, atax, bicg, gesummv, gemm, syrk	743 GB [33 GB]
Microbenchamrks	l1-lat, l1-bw, shd-lat, shd-bw, l2-lat, l2-bw, mem-lat, mem-bw, maxflops	3 GB [94 MB]
CUTLASS [76]	SGEMM with tensor cores WMMA (10 different input) and SGEMM with normal floating point (10 different input)	2.5 TB [125 GB]
Deepbench [77]	GEMM_bench (10 train, 10 inference), CONV_bench (10 train, 10 inference), RNN_bench (10 train, 10 inference)	2.6 TB [130 GB]

techniques and gradual trace creation to support these workloads as future work. We note that without traces, Accel-Sim is still able to execute these apps in execution-driven mode. The total disk size required for all the traces we simulate is 6.2 TB uncompressed (317 GB compressed) for each card generation. Table 3.4 lists the trace size for all the benchmark suites. Generating all 6.2 TB worth of traces takes approximately 48 hours using one GPU.

All the workloads are compiled with CUDA 10, using the compute capability of their respective architecture (i.e. sm_70 in the case of Volta). We run the workloads with the input sizes provided by each benchmark suite. Since deep learning performance is highly sensitive to input size, we run Deepbench with multiple inputs, 10 inputs for training and 10 inputs for inference. For the Volta and Turing cards, we execute two versions of the CUTLASS matrix-multiply apps, one that uses tensor cores and one that uses 32-bit floating point operations. For CUTLASS, the input sizes from DeepBench's matrix multiply kernels are used. Hardware counters are collected using the nvprof [78] and nsight [79] command-line profilers from CUDA 10. We collect the timing data on a per-kernel basis by running the applications several times and averaging the results.

2.4 Modeling Different GPU Generations

To demonstrate Accel-Sim's flexibility, we model four different NVIDIA GPU generations (Kepler, Pascal, Volta and Turing). First, we generate the traces for all the benchmarks using

Table 2.4. Accel-Sim modeling properties across four different GPU generations. SPU: Single-Precision Floating Point Unit. DPU: Double-Precision Unit. SFU: Special Function Unit.

	Kepler TITAN [80]	Pascal TITAN X [81]	Volta QV100 [41]	Turing RTX 2060 [82]
Machine ISA	sm_35	sm_62	sm_70	sm_75
Core Model	shared model	sub_core model	sub_core model	sub_core model
Inst Dispatch/Issue	dual-issue	dual-issue	single-issue	single-issue
Execution units	SPUs, SFUs	SPUs, SFUs, DPUs	SPUs, SFUs, Tensor Cores	SPUs, SFUs, Tensor Cores
L1 Configuration	128B line, 1 bank , 96-way, 128B/cycle, 42 cycles	32B sector, 2 banks, 48-way, 64B/cycle, 80 cycles	32B sector, adaptive, 4 banks, 256-way, 128B/cycle, 28 cycles	32B sector, 4 banks, 96-way, 28 cycles
L2 Configuration	32B sector, 24 banks, 190 cycles	32B sector, 24 banks,238 cycles	32B sector, 64 banks, 212 cycles, IPOLY hashing	32B sector, 24 banks, 226 cycles
DRAM Configuration	GDDR5	GDDR5	HBM	GDDR6

Accel-Sim's tracing tool on each card[1]. Second, we provide the *ISA def* and *HW Def* files for each card from publicly available documents [41], [56] to the Accel-Sim frontend and tuner respectively, as illustrated in Section 2.2. Third, we run our microbenchmark suite and automated tuner on each card to configure the performance model's parameters for the card in question. Fourth, using Accel-Sim's correlation tool, we generate a set of graphs for performance counters to validate and improve correlation accuracy. Table 2.4 shows the machine ISA version and modeling parameters obtained using Accel-Sim's tuner to model each GPU generation. These correlation results drive the development of our performance model and enable us to uncover a number of interesting insights into contemporary hardware in different GPU generations. Three significant aspects are:

Sub-Warp, Sectored Memory Coalescer: We find that in Pascal, Volta and Turing, both the L1 and L2 caches have 32B sectors [83], with 128B cache lines. This sub-line fetching behavior has a significant impact on irregular applications running with the L1 cache enabled. To support an access granularity of 32B into the cache, the memory coalescer operates across sub-warps of eight threads, as opposed to grouping 32 threads into a wide 128B access, as other contemporary GPU simulators do when the L1 cache is enabled. We speculate that the caches have been sectored to: (1) mitigate the memory over-fetching for

[1]↑We use the Volta (sm_70) traces to simulate Turing (sm_75), as NVBit support for Turing is not expected until Summer 2020.

uncoalesced accesses [84], and (2) reduce the cache tag overhead, especially the high power consumption associated with the large content-address memory [85] structures required for 4× more tag comparisons if a 32B line is used as opposed to a 128B with 32B sectors. Our experiments suggest that Kepler does not support such a coalescing mechanism. Accel-Sim's flexibility allows it to model either the legacy Kepler configuration or the Pascal/Volta/Turing configurations with reasonable accuracy.

Improved L1 throughput: The L1 cache in Volta and Pascal is a *streaming cache* [4], [58], which allows many cache misses to be in flight, regardless of how many cache lines are allocated from the unified scratchpad + L1 (in Volta) or the relatively limited 24kB L1 cache (in Pascal). By running a microbenchmark, similar to [43], we evaluate miss-status holding register [86] (MSHR) throughput and find that the number of MSHR entries has increased substantially. Also, we notice that, independent of L1 configured size, the MSHR throughput is the same, even if more of the on-chip SRAM storage is devoted to shared memory.

Write-Aware Cache Design: Using Accel-Sim's detailed correlation plots, our initial experiments found that hardware conserves memory bandwidth much more than expected when running write-intense GPU workloads. To understand the behaviour, we developed an L2 write policy microbenchmark to confirm that the L2 is write-back with a write-allocate policy. Interestingly, we also observed additional behaviour on write misses. Historically, there are two different write allocation policies [83], *fetch-on-write* and *write-validate*. Using Accel-Sim, we find that the L2 cache in all four generations applies a version of *write-validate* that handles the varying granularity of GPU read/write requests. L2 reads have a minimum granularity of 32B, but writes can be sub-32B. When a write to a single byte is received, it writes the byte to the sector, sets a corresponding write bit in a byte-level write mask and sets the sector as valid and modified. When a sector read request is received to a modified sector, it first checks if the sector write-mask is complete, i.e. all the bytes have been written to and the line is fully readable. If so, it reads the sector, avoiding a DRAM access. If the sector is not fully written, it generates a read request for this sector and merges it with the modified bytes. Applying this write policy saves DRAM bandwidth that can be wasted by read-write interference at the DRAM when sub-sector writes are received at the L2 cache.

Table 2.5. Accel-Sim cycle error and correlation factor across four different GPU generations.

	Kepler[80] **TITAN**		**Pascal** [81] **TITAN X**		**Volta** [41] **QV100**		**Turing** [82] **RTX 2060**	
	MAE	Corr.	MAE	Corr.	MAE	Corr.	MAE	Corr.
PTX Exec	28%	0.99	53%	0.97	34%	0.98	32%	0.99
SASS Trace	25%	0.99	30%	0.97	15%	1.00	25%	0.99

To demonstrate Accel-Sim's ability to model different ISAs and hardware generations in both SASS and PTX simulation, Table 2.5 presents the Mean Absolute Error (MAE) and the Karl Pearson Coefficient of dispersion (Correl) for simulated cycles across four different GPU cards. Correl is the ratio of the standard deviation to the mean and indicates how closely the trends in both the simulator data and the hardware data match. Table 2.5 demonstrates that correlation is > 0.97 in all instances and error is $<= 30\%$ for all SASS simulations. The accuracy gained by executing the machine ISA over the virtual ISA can be as much $2\times$, depending on the generation. The one outlier with the greatest error is Pascal in PTX mode. In Pascal, the L1D and texture cache are merged and L1D caching is disabled by default. Without the L1D as a bandwidth filter, L2 throughput is more important in Pascal. The poor instruction scheduling in PTX results in a 53% error that is reduced to 30% when SASS instruction scheduling is used.

2.5 A Detailed Correlation Analysis

In this section, we perform a quantitative analysis to demonstrate how our new performance model can be used to simulate an NVIDIA Quadro V100 [4]. We compare Accel-Sim against the state-of-the-art NVIDIA simulation model from GPGPU-Sim 3.x [40], [46]. The GPGPU-sim 3.x model is a best-effort attempt to model the Volta card using GPGPU-Sim 3.x without the modeling updates described in this work. Table 2.6 details the configuration parameters used to model the architecture. The left column of the table details the configuration of GPGPU-sim 3.x, obtained by scaling resources. The right column lists changes made possible using Accel-Sim and our updated GPGPU-Sim 4.0 performance model. For the sake of brevity, we refer to GPGPU-Sim 3.x as 'GPGPU-Sim' and Accel-Sim with GPGPU-Sim

Table 2.6. Volta GPU configuration for GPGPU-Sim 3.x vs Accel-Sim. INT: Integer operation unit.

	GPGPU-Sim 3.x	**Accel-Sim (SASS + GPGPU-Sim 4.0)**
Front End	Execution-driven PTX 2.0	**Trace-driven SASS 7.0**
#SMs	80	80
SM Configuration	Warps per SM = 64, #Schedulers per SM = 4, #Warps per Sched = 16, RF per SM = 64 KB	**+Subcore model, warp scheduler isolation**
#Exec Units	4 SPUs, 4 SFUs, 4 WMMA Tensors cores [57]	**+ 4 DPUs, 4 INTs, 8 HMMA Tensor cores**
Memory Unit	Fermi coalescer (32 thread coalescer)	**sub-warp (8 thread) coalescer + Fair memory issue**
Shared Memory	Programmable-specified up to 96 KB	**Adaptive (up to 96 KB), latency = 20 cycles**
L1 cache	32KB, 128B line, 4 sets, write-evict	**+32B sector, adaptive cache (up to 128 KB), Streaming cache, banked, latency = 28 cycles**
L2 cache	6 MB, 64 banks, 128B line, 32 sets, Write Back, Naive write policy, LRU, latency=100 cycles	**+32B sector, subsector write policy, memory copy engine model, +pseudo-random irreducible polynomial L2 hashing with IPOLY(67)**
Interconnection	80x64 crossbar, 32B flit	
Memory Model	GDDR5 model, BW=850 GB/sec, latency=100ns	**HBM Model, dual-bus interface, Read/Write buffers, advanced bank indexing**

Table 2.7. Error and correlation rates of GPGPU-Sim 3.x versus Accel-Sim when modeling an NVIDIA Volta.

Statistic	Error		Correlation	
	GPGPU-Sim 3.x	Accel-Sim	GPGPU-Sim 3.x	Accel-Sim
L1 Reqs	NRMSE=3.04	NRMSE=0.00	0.99	1.00
L1 Hit Ratio	NRMSE=1.04	NRMSE=0.76	0.69	0.87
Occupancy	NRMSE=0.12	NRMSE=0.13	0.99	0.99
L2 Reads	NRMSE=2.67	NRMSE=0.03	0.95	1.00
L2 Read Hits	NRMSE=3.24	NRMSE=0.47	0.82	0.99
DRAM Reads	NRMSE=5.69	NRMSE=0.92	0.89	0.96
Instructions	MAE=27%	MAE=1%	0.87	1.00
Execution Cycles	MAE=94%	MAE=15%	0.87	1.00

4.0 as 'Accel-Sim'. Note that both simulators are capable of executing NVIDIA's Tensor Core instructions. GPGPU-sim uses the PTX-level WMMA tensor core instruction. Based on the analysis from Raihan et al. [57], Accel-Sim models the fine-grained, SASS-level HMMA instruction. Since Accel-Sim is capable of both SASS-trace and PTX-execution driven simulation, we refer to them as Accel-Sim and Accel-Sim [PTX Mode] respectively.

In this section, we demonstrate the accuracy of our more flexible performance model and the effect of being able to simulate SASS traces. The correlation figures presented throughout this section plot hardware results from the NVIDIA profilers nvprof [78] and nsight-cli [79]

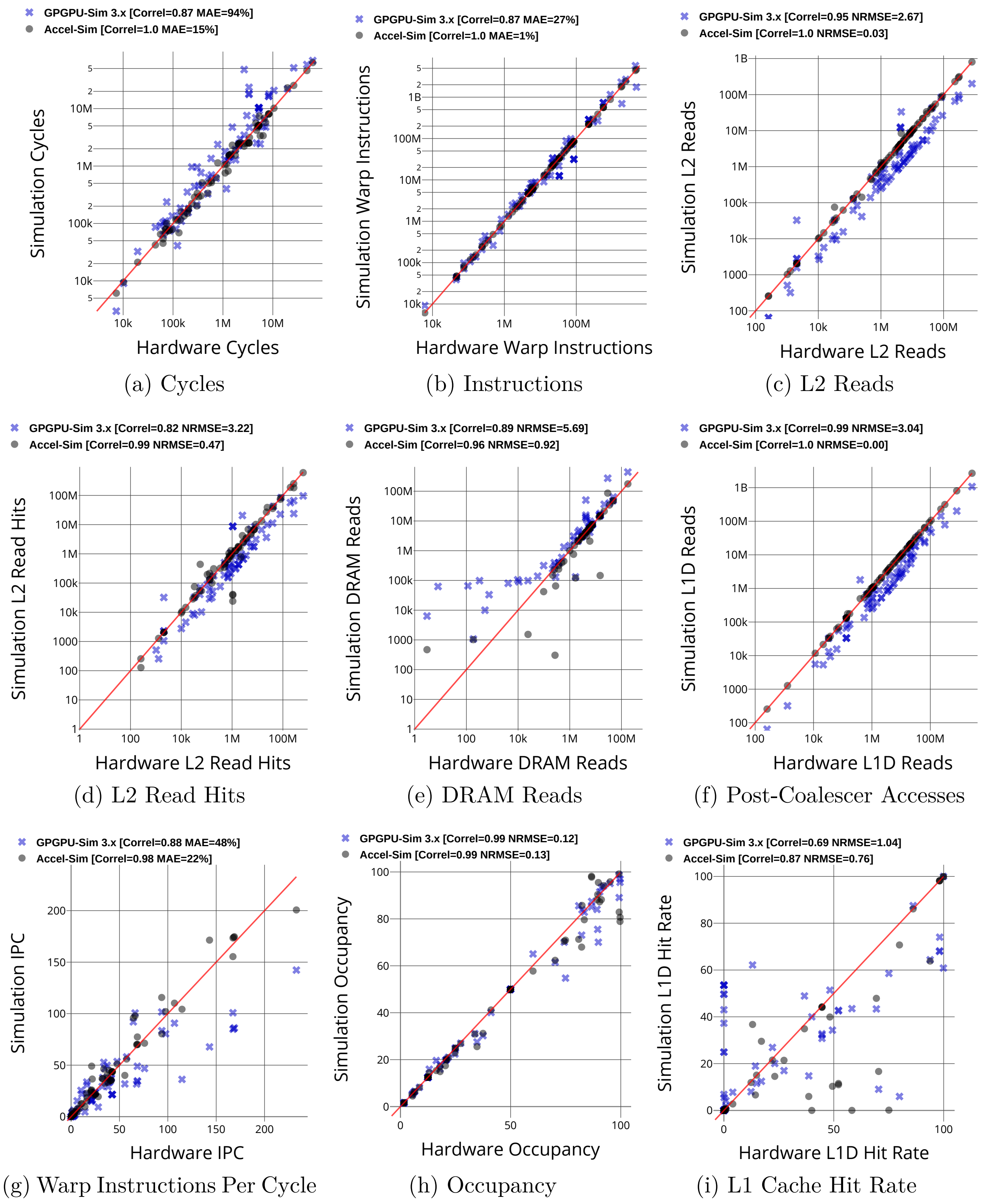


Figure 2.4. Correlation of key metrics from Accel-Sim versus GPGPU-Sim 3.x over the 80 workloads. These workloads are listed in Tables 2.8 and 2.9.

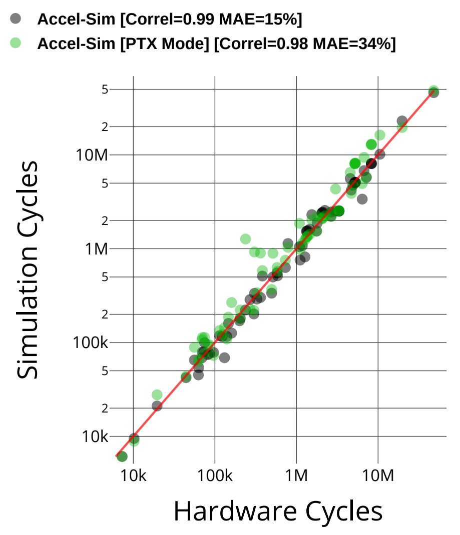


Figure 2.5. Cycle correlation of Accel-Sim in execution-driven PTX mode versus Accel-Sim in trace-driven SASS mode.

on the x-axis and the simulation results on the y-axis. Depending on the statistic, there are up to 80 datapoints on each graph (80 applications, without the Deepbench workloads). The blue **x** represents the GPGPU-Sim result and the circles represent Accel-Sim. Each plot also lists the Correl and error for the statistic. We use two different aggregation techniques for error. Intuitively, the Mean Absolute Error (MAE) makes the most sense since it provides a percentage error and is resistant to outliers. However, calculating MAE is not possible if the reference counter is zero and creates the appearance of massive error when small values deviate. As a result, we cannot use the MAE for counters other than the number of cycles, instructions executed and Instructions Per Cycle (IPC), which will always be non-zero values. For the remaining counters, we use the Normalized Root Mean Squared Error (NRMSE). Table 2.7 summarizes the error and correlation co-efficients for GPGPU-Sim versus Accel-Sim, over a number of important performance metrics.

2.5.1 Overall System Correlation

Figure 2.4 details a direct comparison of GPGPU-Sim to Accel-Sim over a set of key simulation metrics. Figure 2.4a plots the final number of cycles reported for each application. Rounded to 2 significant digits, Accel-Sim has a 1.00 correlation with hardware, and a

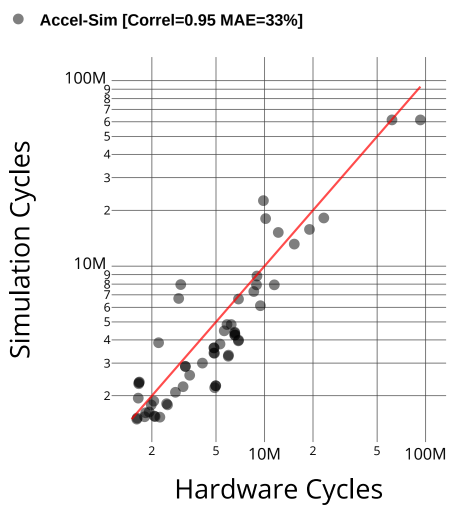


Figure 2.6. Accel-Sim cycle correlation when executing deepbench workloads, which cannot be executed by GPGPU-Sim 3.x or Accel-Sim [PTX Mode]. 60 workloads (comprised of 11,440 kernel instances).

15% mean absolute error. In contrast, GPGPU-Sim has a correlation of 0.87 and 94% error. Generally, GPGPU-sim's values are above the x=y line, meaning the simulator is over-estimating the number of cycles for the workload. GPGPU-Sim is not modeling the performance enhancements made to the memory system, which Accel-Sim does.

Accel-Sim's increased accuracy can be attributed to two factors: the machine ISA representation and the updated performance model. To understand the effect of the instruction set, Figure 2.4b plots the number of warp instructions executed. Since GPGPU-Sim operates on PTX, the correlation is relatively low and shows a 27% error. In contrast, Accel-Sim has a 1.0 correlation and a 1% error. Accel-Sim's small error occurs because some applications execute a non-deterministic number of instructions in hardware. Figure 2.4b demonstrates that PTX generally executes a reasonable number of instructions. However, the ordering of the instructions is significantly different in SASS, resulting in better instruction-level parallelism. To quantify the difference between PTX and SASS more clearly, Figure 2.5 shows the cycle correlation when using Accel-Sim [PTX Mode] versus Accel-Sim. Figure 2.5 allows us to examine the effect of SASS versus PTX on the same performance model. On average, simulating SASS improves the mean absolute error by 2×, reducing the error from 34% to 15%. To provide additional insight on an app-by-app basis, Tables 2.8 and 2.9 enumerates

Table 2.8. Cycle mean absolute error per workload in GPGPU-sim 3.x, Accel-Sim [PTX Mode] and Accel-Sim for SDK, Rodinia and Parboil

Workload	HW cycles	GPGPU-Sim 3.x Error (%)	Accel-Sim [PTX] Error (%)	Accel-Sim Error (%)
CUDA SDK				
blk	10214	-13	-11	-7
conv	327868	3	30	-11
fast-walsh-transform	731574	35	-6	4
merge-sort	52905	17	-15	2
scalar-prod	300811	-30	-30	-32
sorting-networks	85713	10	-13	-10
transpose	97339	-2	12	-19
vector-add	140161	-22	32	-17
scan	2659445	29	-16	-16
histo	61085	21	21	-9
CUDA SDK MAE	-	**13**	**16**	**9**
Rodinia Suite				
b+tree	167146	-7	-8	-19
backprop	63456	0	4	-14
bfs	1107676	61	-11	-31
dwt2d	35246	90	51	-3
hotspot	56101	63	58	16
lud	710043	9	48	-9
myocyte	4561278	51	42	22
neddle	3016000	151	43	-16
srad-v1	6031393	-7	42	0
sc	699593	82	-15	-8
kmeans	1437879	300	300	63
nn	7305	-59	-40	-11
lava-md	5017353	20	12	8
Rodinia MAE	-	**32**	**28**	**8**
Parboil Suite				
bfs	6396950	-23	-22	-47
histo	5590035	-56	-59	-39
mri-q	299165	220	207	9
sad	131272	4	9	-47
sgemm	362333	111	148	-16
spmv	3405773	38	-8	-7
stencil	4698755	-46	-17	-10
cutcp	5321758	75	75	39
Parboil MAE	-	**45**	**37**	**20**

the per-application cycle error for GPGPU-Sim, Accel-Sim [PTX Mode] and Accel-Sim. In many of the CUTLASS *sgemm* and Rodinia workloads, the error is significantly reduced by moving to SASS-based simulation. In these workloads, capturing the improved instruction scheduling in SASS is critical to simulator accuracy.

In addition to examining SASS versus PTX, Tables 2.8 and 2.9 demonstrate some interesting trends across benchmark suites. In the CUDA SDK, many of the workloads are streaming and sensitive to DRAM bandwidth. As such, our detailed HBM model and SASS representation reduces the error in these workloads, providing relatively low error. In the

Rodinia suite, which includes a number of workloads that have cache-locality and more diverse, irregular behaviour, Accel-Sim's detailed cache model, combined with SASS instruction scheduling helps to reduce the error. Although Accel-Sim's overall correlation and accuracy are greatly improved, there are still several benchmarks with a high error. Further investigation reveals that correctly modeling the L2 bandwidth is critical in the Rodinia workloads. Our IPOLY indexing helps to alleviate many L2 bank conflicts, however it does not exactly match the hardware's indexing function, as described in Section 2.5.2. The Polybench workloads are cache capacity sensitive [87]. GPGPU-sim employs a fixed capacity of 32KB in the L1, whereas the adaptive, sectored cache in Accel-Sim allocates the whole 128KB of Volta's unified on-chip memory to the L1D. The 4× greater cache capacity and more accurate 32B fetch granularity reduces the error in *atax, bicg, gesummv, syrk* and *mvt* from 400% to 20% on average.

For the CUTLASS *sgemm* workloads (without Tensor Core instructions) the streaming L1 cache, advanced L2 hashing and support for SASS all contribute to reducing the error in Accel-Sim to less than 5%. Interestingly, Accel-Sim's error for CUTLASS *gemm-wmma* is higher than *sgemm*. In addition, GPGPU-Sim's error is generally much lower on *gemm-wmma* than on *sgemm*. We find that the abstract WMMA PTX instructions are a good approximation for the SASS instructions. For some input sizes, the abstract instructions are an even better representation than Accel-Sim's SASS model for HMMA. In these instance, Accel-Sim [PTX Mode] demonstrates nearly zero cycle error, since it benefits from both the memory system updates and the accuracy of the abstract WMMA instructions.

We also plot the attained occupancy (Figure 2.4h) and IPC (Figure 2.4g). The occupancy data tracks well for both GPGPU-Sim and Accel-Sim, both models correctly estimate the initial occupancy, deviation from hardware occurs in imbalanced kernels where occupancy falls off at different rates. The purpose of correlating IPC is to demonstrate how well the simulator handles different rates of work. In order to avoid negatively biasing GPGPU-Sim (which uses a different instruction set than both Accel-Sim and hardware), we use the machine instruction count as the numerator to calculate the IPC in both simulators. Figure 2.4g follows from Figure 2.4a, where Accel-Sim shows better correlation and less error. GPGPU-Sim's IPC error is less than its cycle error. Most of GPGPU-Sim's error occurs in long running apps, where the gap in cycles becomes orders of magnitude larger.

Finally, Figure 2.6 plots the Deepbench workloads that make use of the closed-source cuDNN and cuBLAS libraries, that include hand-tuned SASS code for which there is no PTX representation. Only Accel-Sim is able to simulate these workloads. Generally, Accel-Sim is well correlated, although the overall error is higher than for the other workloads. Further inspection reveals that the Deepbench workloads make extensive use of texture memory and local loads, which could require additional modeling to improve their accuracy. For hardware occupancy, Accel-sim achieves 0.9 correlation and 0.16 NRMSE on the Deepbench workloads. Low occupancy in RNN workloads has been observed by prior work [88] in silicon GPUs, which we also observe in Accel-Sim. Accel-Sim's support of NVIDIA's hand-tuned libraries enables new architecture research into the performance of RNNs and other deep learning networks on GPUs, which is not possible in any other simulation framework.

2.5.2 Achieved Bandwidth Correlation

GPU workloads are highly sensitive to cache and memory bandwidth, making them important factors for modeling GPUs accurately. Significant effort has been expended to design contemporary GPUs to achieve the highest possible bandwidth. To measure how much memory bandwidth can be saturated, we design a set of microbenchmarks to saturate the memory bandwidth of the L1, L2 and main memory. Figure 2.7 depicts the cache and memory bandwidth attained for in the Volta card, normalized to the theoretical bandwidth. In Volta [4], [45], the theoretical bandwidth for both simulators have been configured to supply, at most, the theoretical bandwidth of each component.

As shown in Figure 2.7, GPGPU-Sim is only able to achieve 33% of the theoretical L1 bandwidth, whereas the Accel-Sim's streaming and banked L1 cache is able to attain 85%, within 10% error of the attained hardware bandwidth. In L2 cache, thanks to our advanced Galios-based random hashing and accurate interconnect bandwidth, Accel-Sim mitigates L2 partition camping, which has been exacerbated in contemporary GPUs with the introduction of HBM [63], [64], and comes within 22% of hardware. We believe that the

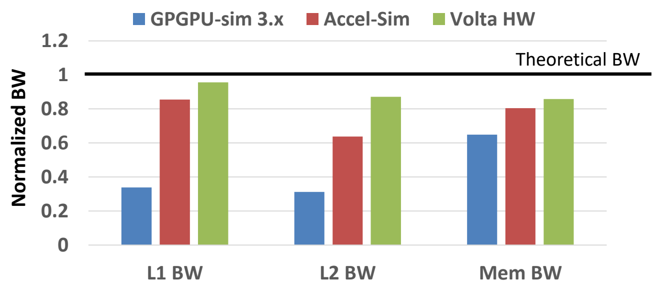


Figure 2.7. Measured cache and memory bandwidth for Accel-sim, GPGPU-sim 3.x and Volta hardware .

hashing mechanism used in contemporary GPUs is highly tuned, and eliminating the last 22% error would require extensive reverse-engineering. We leave improving the L2 bandwidth for future work. For memory bandwidth, our detailed HBM model with a dual-bus interface, the random partition indexing and write buffers employed in Accel-Sim achieves 82% of the theoretical memory bandwidth and very close to the 85% attained in hardware. In the GPGPU-Sim, read/write interference, low bank-group level parallelism and naive L2 write policy waste memory bandwidth leading it to attain only 62% bandwidth.

2.5.3 L1 Cache Correlation

Figure 2.4f plots correlation for the post-coalescer accesses received by the L1 cache. As Figure 2.4f shows, Accel-Sim has a correlation co-efficient of 1.00 correlation and no error. This is primarily due to the implementation of the subwarp coalescer and the fact that the L1 cache is sectored. Figure 2.4i plots the corresponding hit rate for the L1D cache. We note that this statistic is particularly hard to correlate, given that the warp scheduling policy, cache replacement policy and hit/miss accounting decisions in the profiler can skew the results. For example, through microbenchmarking, we determined that the hardware profiler will count a sector miss on a cache line whose tag is already present as a hit, but will still send the sector access down to the L2. Effectively, the profiler appears to be counting 128B line cache tag hits, even if the sector is missed. Even with these variables, Accel-sim's cache model achieves only 0.77 NRMSE and a correlation co-efficient of 0.87. In the apps where pervasive error still exists, we believe more detailed reverse engineering of the warp scheduling policy and replacement policy will help.

GPGPU-sim have a high correlation, but an NRMSE of 3.06 in L1D read accesses. GPGPU-sim does not model a sectored cache, thus one 128B access will count as four sector accesses in the hardware. Completely divergent applications show consistent behaviour with hardware since 32 unique sectors are generated in hardware and 32 unique cache lines are generated in GPGPU-sim.

2.5.4 L2 Cache Correlation

Although all L1D read misses are sent to L2 cache, Accel-Sim's L2 reads has higher correlation and less error compared to the L1 hit rate. This result confirms our observation that the profiler's L1 hit/miss accounting decisions are different from our assumption. As shown in Figure 2.4c, Accel-sim exhibits 1.00 correlation and normalized error reaching only 0.03. We also collected information on write transactions, and noticed similar behaviour.

We attempted to correlate the L2 cache read hit rate with simulation results, but found that the profiler gives inconsistent results, some of which are greater than 100%. So instead, we correlated with the number of L2 hits, which gave more grounded readings. Figure 2.4d presents the L2 read hit correlation. Overall, Accel-sim achieves 0.99 correlation and 0.47 normalized error. Again, at the L2, scheduling effects from the order in which memory accesses reach the L2 will have an effect on it's hit rate.

2.5.5 DRAM Correlation

Finally, Figure 2.4e plots the correlation of DRAM read accesses. The NRMSE is 0.93 and has a 0.96 correlation co-efficient. The DRAM read correlation remains high and is more accurate for larger kernels. However, Accel-sim shows a high level of error for smaller workloads.

The results in GPGPU-sim are significantly worse than those in Accel-Sim. Interestingly, the reason for the massive error in the GPGPU-sim is due to its per-cacheline fetch on write policy. The end result of that policy was that every write to the writeback L2 fetched 4

32-byte memory values from DRAM, which is why the DRAM reads of the GPGPU-sim are consistently overestimated. Accel-sim has eliminated this problem by sectoring the cache and implementing a more accurate write policy, as explained in Section 2.4.

2.6 Case Studies

This section presents the effect of memory system simulation detail on architectural design decisions. A less detailed architecture model may lead to unrealistic issues or incorrect conclusions that are not relevant to state-of-the-art designs already being used in industry. This section demonstrates how our more accurate GPU model removes the well-studied bottleneck of cache throughput and opens up new research opportunities in chip-wide memory issues.

L1 cache throughput: The L1 cache throughput bottleneck for GPUs has been well explored in literature using cache bypassing and warp throttling techniques [89]–[93]. Due to massive multithreading, GPGPU L1 caches can suffer from severe resource contention (e.g. MSHRs and cache line allocation [89]). However, contemporary GPUs are designed such that the L1 cache is not a throughput bottleneck. Modern GPUs employ advanced architecture techniques to improve L1 cache throughput, such as sectored, banked, adaptive and streaming L1 caches. These techniques shift the memory bottleneck from the L1 cache to the lower levels of the memory hierarchy. Figure 2.8 plots the L1 cache reservation fails per kilo cycles for cache sensitive workloads for GPGPU-sim vs Accel-Sim. The new L1 model eliminates reservation fails and the L1 cache is no longer a throughput bottleneck. Consequentially, any design aimed at mitigating this L1 throughput bottleneck is less effective on a contemporary GPU, which is reflected in our new performance model.

Memory scheduling sensitivity: Throughput-oriented GPGPU workloads are often sensitive to memory bandwidth. Thus, the DRAM memory access scheduler plays an important role to efficiently utilize bandwidth and improve the performance of memory-intensive workloads (i.e. those that utilize $> 50\%$ of DRAM bandwidth). Figure 2.9 shows the sensitivity of memory-intensive GPGPU workloads for two memory scheduling policies: the naïve first-come first-serve scheduling (FCFS) and the advanced out-of-order First-row-ready first-

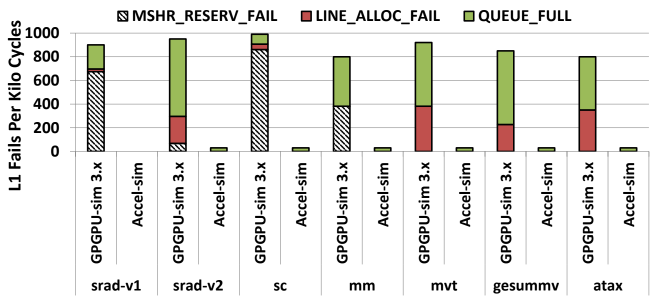


Figure 2.8. L1 cache reservation fails per kilo cycles for cache sensitive workloads in both Accel-Sim and GPGPU-sim 3.x.

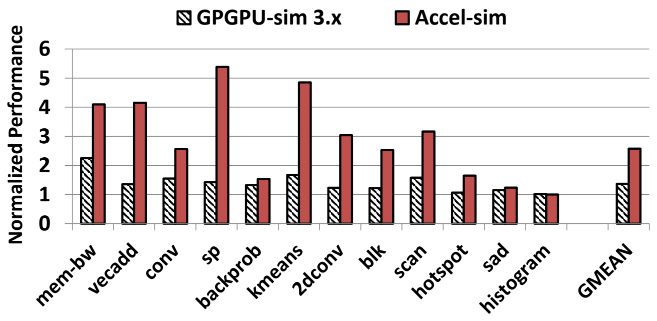


Figure 2.9. FR_FCFS performance normalized to the FCFS in Accel-Sim and GPGPU-sim 3.x for memory intensive workloads in Volta.

come first-serve scheduling (FR_FCFS) [94]. FR_FCFS performance is normalized to FCFS in both Accel-Sim and GPGPU-sim to demonstrate the performance improvement from more advanced scheduling. In the GPGPU-Sim, some workloads are insensitive or show little difference between the two scheduling policies. On average, applying FR_FCFS in GPGPU-sim increases performance by 20%.

In Accel-Sim, the memory scheduling policy is more critical. Applying FR_FCFS to Accel-Sim improves performance by 2.5× on average. More detailed coalescing rules, improved on-chip cache throughput, advanced L2 cache interleaving and write allocation policies increase the likelihood of losing page locality, which must be recaptured by out-of-order memory access scheduling. Further, with the new features and capabilities of the HBM such as dual-bus interface [67], pseudo-independent accesses [95] and per-bank refresh command [66], memory scheduling and the interaction with the L2 cache indexing will become a more critical issue to investigate. This experiment demonstrates how accurately modeling contemporary

GPU hardware reveals performance issues obscured using a less detailed simulation. We believe Accel-Sim opens up a rich new design space in system-level GPU research where core-level memory issues are alleviated and there is increased sensitivity to chip-wide memory issues.

2.7 Summary

This chapter introduces three innovative mechanisms to help solve the problem of keeping simulators up-to-date with contemporary designs. We propose a novel, ambidextrous GPU frontend that translates both functionally executed virtual instructions and machine instruction traces into an ISAindependent format for simulation. Using the flexible frontend, a detailed performance model is developed, which has the ability to model contemporary GPUs and execute proprietary binaries that no other open-source simulator can. We believe that Accel-Sim is the most extensively validated open-source GPU simulation framework to date. We demonstrate that Accel-Sim decreases cycle error from 94simulation to 15Accel-Sim's ISA-independent performance model opens up opportunities to simulate cards from a wide variety of vendors. We believe that Accel-Sim will reduce the accuracy gap between industrial and academic simulators on an ongoing basis, increasing the potential impact of accelerator research.

Table 2.9. Cycle mean absolute error per workload in GPGPU-sim 3.x, Accel-Sim [PTX Mode] and Accel-Sim for Polybench, Microbenchmark, and CUTLASS

Polybench Suite				
2dconv	269298	62	6	-15
2mm	62994676	8	0	0
3dconv	1788022	-26	-17	3
3mm	1766299	9	-12	-12
atax	3322009	432	-23	-23
bicg	3330876	604	-24	-23
gemm	587160	19	-12	-1
gesummv	2661367	1700	-7	-7
mvt	3323425	445	-24	-23
syrk	15668564	658	-7	-7
Polybench MAE	-	**111**	**6**	**4**
Microbenchmark Suite				
maxflops	73152	39	3	3
l1-bw	578838	134	9	-4
l1-bw-unroll	237705	304	433	-6
l1-lat	1177511	-65	-2	-8
shared-bw	4921085	-1	-3	-3
shared-lat	70231	6	12	12
l2-bw	786881	202	31	44
l2-lat	7184779	-33	-18	-19
mem-bw	382171	21	15	16
mem-lat	2389456	-71	10	-16
Microbenchmark MAE	-	**37**	**14**	**9**
CUTLASS GEMM wmma Suite				
wmma-2560x128x2560	1455597	11	-1	10
wmma-2560x16x2560	1325345	7	0	15
wmma-2560x32x2560	1338720	6	0	14
wmma-2560x64x2560	1376862	4	0	13
wmma-2560x7000x2560	19717874	12	0	16
wmma-4096x128x4096	2231763	32	0	14
wmma-4096x16x4096	2055799	10	1	17
wmma-4096x32x4096	2060767	12	1	17
wmma-4096x64x4096	2103894	17	2	16
wmma-4096x7000x4096	48103472	21	0	-4
CUTLASS wmma MAE	-	**11**	**0.5**	**12**
CUTLASS sgemm Suite				
sgemm-2560x1024x2560	10525540	96	54	-3
sgemm-2560x128x2560	5272700	97	54	-3
sgemm-2560x16x2560	5199087	97	55	-2
sgemm-2560x2560x2560	26224856	95	55	-3
sgemm-2560x32x2560	5193979	97	55	-2
sgemm-2560x512x2560	5289253	97	54	-3
sgemm-2560x64x2560	5187370	98	55	-2
sgemm-4096x128x4096	8352408	113	55	-3
sgemm-4096x16x4096	8266794	97	55	-2
sgemm-4096x64x4096	8240064	100	56	-1
CUTLASS sgemm MAE	-	**98**	**54**	**2**

3. LADM: A SCALABLE AND TRANSPARENT MULTI-GPU SYSTEM FOR DL AND HPC WORKLOADS

GPU accelerated workloads are commonly used in deep learning and exascale high performance computing (HPC) systems [5], [6]. These workloads exhibit high levels of implicit parallelism, which enables performance scalability, but only if GPUs can continue to scale their hardware resources. Over the past decade, GPUs have more than quadrupled the number of Streaming Multiprocessors (SMs) in their designs, while simultaneously increasing their on-chip transistors by an order of magnitude. Prior work by Arunkumar et al. [3] demonstrates linear performance scalability if GPU resources (SMs, SM-interconnect bandwidth, registers, caches, and DRAM bandwidth) are able to scale proportionally. However, building a GPU with hundreds of SMs in a single monolithic GPU die will not be possible due to low manufacturing yields and the high cost of building large chips at small technology nodes [3], [7].

To overcome these problems and enable continuous performance scaling as Moore's law slows [8], [9], researchers have proposed increasing GPU transistor count by aggregating multiple GPUs together (as a single logical GPU) as well as disaggregating single-GPUs into scalable multi-chip-modules [3], [10], [96], [97]. Compared to single-chip systems, chiplet based architectures [1] are desirable because they provide a larger aggregate chip perimeter for I/O, enabling a higher number of DRAM interfaces to be connected to the system and thus scale memory bandwidth and capacity alongside compute resources [3], [5], [98]–[101].

Future chiplet-based designs will be limited by the size of silicon interposers or the short haul, high bandwidth interconnects needed to traverse a small printed circuit board. Compared to chiplets, multiple GPUs are more easily combined into large coordinated systems but suffer from lower inter-GPU bandwidth, which increases the NUMA performance penalty. As shown in Figure 3.1, these approaches are complimentary and it is likely that both will be employed in future systems with hierarchical interconnects to create a massive logical GPU.

Architecture and runtime systems must coordinate to maintain the existing single-GPU programming model and support transparent scaling for current CUDA programs. The goal

[1]↑In this thesis, chiplet and multi-chip-module (MCM) are used interchangeably.

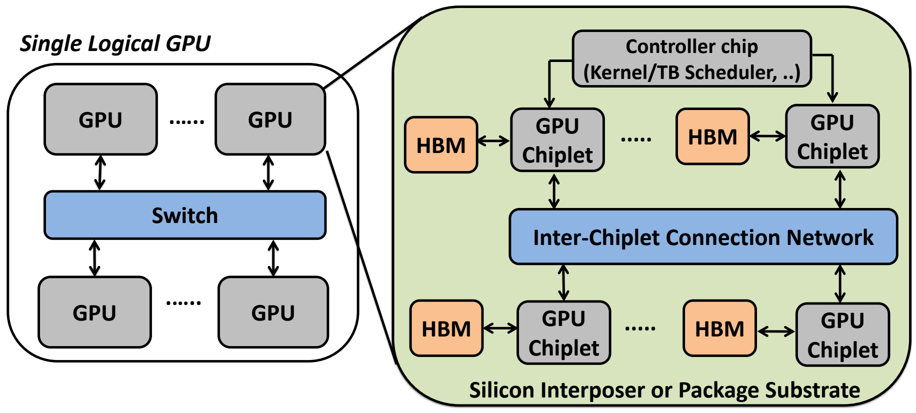


Figure 3.1. Future massive logical GPU containing multiple discrete GPUs, which are themselves composed of chiplets in a hierarchical interconnect.

is to create a *single programmer-visible GPU* that may be comprised of hierarchical locality domains. Maintaining this illusion enables rapid software development on small local GPU resources while allowing scalable performance on larger and more complex GPU systems. Transparently overcoming locality effects will be a challenging problem for GPUs over the next decade. Such an extreme NUMA scale requires new techniques to place pages, cache data, and schedule the many thousands of threads managed in such systems.

Recent work on static analysis for transparent multi-GPU programs, CODA [102], is a step in the right direction. Using the compiler to perform index analysis, CODA calculates the width of the data accessed by one threadblock and ensures that threadblocks and the data they access are placed on the same GPU for the locality types they can identify. However, a more robust analysis of the code is required to exploit different GPU access patterns on hierarchical GPUs. In this work, we deconstruct the accesses patterns observed across a diverse set of GPU applications and detail which patterns are captured by recent state-of-the-art NUMA-GPU mechanisms and those that remain unexploited. We show that many of the previously unexploited patterns can be successfully detected by static analysis, which we use to drive data placement, caching, and thread scheduling decisions in our Locality-Aware Data Management (LADM) system.

Static index analysis has been extensively used in sequential code to perform affine loop transformations, eliminate data dependencies, and partition work for automatic parallelization [103], [104]. In many ways, a static analysis of a GPU program is more straightforward than a sequential one, as the parallelism in CUDA programs is inherent to the programming model. A parallel GPU program can be transformed into a sequential program by converting the threadblock and grid dimensions into loop index variables on data-parallel outer loops. Once this transformation is made, any sequential program analysis can be applied to the GPU code. However, it is less obvious how the nature of the hierarchical programming model (i.e., threadblocks and kernels) can be combined with sequential locality analysis to map schedulable chunks of work (i.e., threads within the same threadblock) to data structure accesses. To tackle this issue, we introduce the concept of datablock locality analysis that maps each threadblock in a kernel to chunks of data we predict it will access.

Fundamentally, the programming model for GPUs is different than for CPUs. Due to their massively-multithreaded nature, GPU programs are composed of many fine-grained threads, where each individual thread exhibits little spatial or temporal locality to global memory. This, combined with the expressiveness of thread IDs in the CUDA programming model creates both a new challenge and an interesting opportunity to apply static analysis for NUMA data placement and thread scheduling.

3.1 Background

Figure 3.1 depicts what next-generation exascale GPU compute accelerators may look like in the future. Within a single GPU, monolithic GPU dies will be subdivided into disaggregated chiplets, where each chiplet is composed of a group of SMs associated with its own local High Bandwidth Memory (HBM) and hardware thread block scheduler. Several different ways to connect these chiplets have been proposed. Interposer-based through-silicon vias (TSVs), similar to those proposed in AMD's future exascale node [5], [97], high rate signaling through organic substrate-based connections similar to NVIDIA's Ground Reference Signaling (GRS) [3], [105], [106], Intel's Embedded Multi-die Interconnect Bridge (EMIB) [107] or waferscale integration using Silicon Interconnection Fabric (Si-IF) [96], [108] are all possible solutions. While these links may provide high enough bandwidth to alleviate the NUMA-GPU performance penalty [5], such solutions are likely to be expensive and hard to manufacture. These same technologies could be conservatively applied to provide cost-effective, bandwidth restricted interconnections [3]. Architectural and software techniques are necessary to reduce off-chiplet traffic and mitigate the performance loss due to bandwidth constraints. While reducing off-chiplet traffic across exotic high-speed connections may not lead to performance improvement, LADM still improves overall energy efficiency by minimizing data movement among the chiplets [101].

NUMA-GPU designs will not only exist within on-package solutions. With the arrival of high bandwidth switch-connected GPUs such as NVIDIA's DGX-2 and NVLink [109], [110] interconnect, aggregating multiple discrete GPUs into a large virtual GPU is now being considered [10]. Because these GPUs may operate as both individual GPUs and in aggregate (as a single GPU), this aggregation must be done with more limited hardware support, primarily by the GPU runtime software. In addition, the type of hierarchical NUMA present in Figure 3.1 must be accounted for both page placement and threadblock scheduling. Previous, hierarchy-oblivious approaches [3], [10], [102] to NUMA-GPU should be applied recursively, accounting for the fact that chiplets on the same discrete GPU will have greater peer-to-peer bandwidth than chiplets that reside on different GPUs.

3.1.1 NUMA Locality in CPUs vs GPUs

Parallel programming on NUMA multi-processor and on-chip multi-core CPU systems is a well studied problem. Many proposals attempt to minimize NUMA memory access latency transparently through software memory allocation and migration policies[111]–[115] or thread-to-core mapping [116]–[119] techniques. Most of these works are reactive solutions, wherein they detect locality and congestion at runtime, then they perform page migration, replication and thread-clustering based on runtime observations. Although reactive systems can be applied to GPUs, they introduce a substantial performance penalty that can outweigh the benefits. For example, data replication increases memory capacity pressure, which is a scarce resource in contemporary GPUs [120]. First-touch page placement policy can reduce performance significantly, stalling SMs for 20-50 microseconds [121]. Furthermore, the sheer number of threads in flight makes reactive work re-distribution intractable, and the cost of

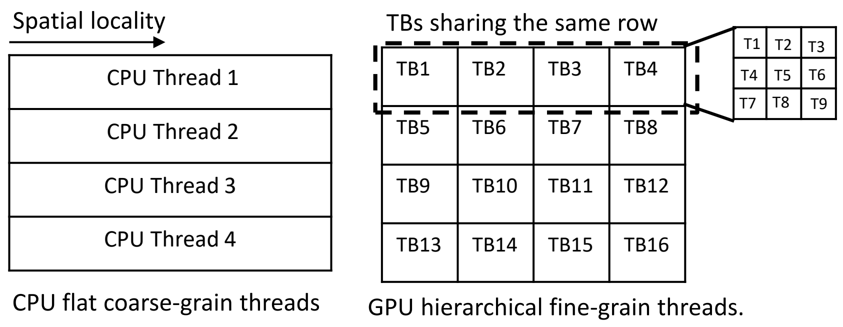


Figure 3.2. OpenMP vs CUDA thread mapping for *sgemm* [74].

page migration in bandwidth-limited GPU workloads is high [122], [123]. These all motivate a *proactive*, prediction-based solution based on static program analysis.

The GPU programming model introduces new challenges in the design space for NUMA systems that did not exist in traditional NUMA-based multi-processor systems. Since GPUs are programmed using a huge number of threads, the work done by each individual thread is small, resulting in far more thread scheduling decisions. To manage all these threads, they are grouped into a multi-dimensional grid of threadblocks, where each block operates on one multi-dimensional chunk of the data structure. This is in contrast to the coarse-grain nature of CPU threads, where far fewer threads do much more work each. Figure 3.2 illustrates how threads in CPUs and GPUs typically access a row-based data structure with an example from the Parboil benchmark suite [74]. In the coarse-grained CPU case, each thread has significant spatial locality and static index analysis of the single-threaded code can easily determine the access pattern of each thread. In the fine-grained GPU case, the same per-thread analysis can be applied. However, the reach of each individual thread is minimal, as each thread will access very few (or even just one) elements in the row. In order to capture the locality pattern in GPUs, an inter-thread analysis must be performed, to account for both the hierarchy of the grid (i.e. the presence of threadblocks) and the dimensionality of the thread grid. This type of inter-thread analysis is what we propose in LADM.

In addition, there is little intra-thread locality in highly-optimized GPU applications with regular access patterns. Instead of repeatedly accessing values on the same cache line, GPU

Table 3.1. LADM vs state-of-the-art techniques

	Batch+FT [3]	Kernel-wide [10]	CODA [102]	LD / TB clustring [50], [124], [125]	LADM
Page placement policy	First-Touch	Kernel-wide chunks	Sub-page round robin	Hand-tuned APIs	LASP
Threadblock scheduling policy	Static batched round robin	Kernel-wide chunks	Alignment-aware batched round robin	Hand-tuned APIs	LASP
Page alignment	✗	✓	✓	✓	✓
Threadblock-stride aware	✓	✗	✗	✓	✓
Row sharing	✗	✓	✗	✓	✓
Col sharing	✗	✗	✗	✓	✓
Adjacent locality (stencil)	✗	✓	✗	✓	✓
Intra-thread loc	✓	✗	✓	✗	✓
Input size aware	✗	✗	✗	✓	✓
Overhead	+First-touch page faulting	-	+Hardware for sub-pages	+APIs	-
Transparency	✓	✓	✓	✗	✓
Hierarchical-aware	✗	✗	✗	✗	✓

programs typically access values on the same line in different coalesced threads. Optimized GPU programs also make extensive use of a scratchpad memory, which effectively prevents a large portion of global data from being accessed more than once. The end result is that there is very little global data reuse in GPU programs, making the initial decision on where a page should be placed extremely important, since temporal locality in upper-level caches is rare.

3.1.2 Existing NUMA-GPU Optimizations

In this section, we qualitatively and quantitatively study state-of-the-art NUMA-GPU page placement and threadblock scheduling techniques for both MCM [3] and Multi-GPU [10], [102], [124] configurations, teasing out the fundamental properties they exploit and highlighting opportunities they miss.

The first work on chiplet-based GPUs by Arunkumar et al. [3] optimizes per-chiplet locality through a synergistic approach of statically batching threadblocks and performing a reactive first-touch (Batch+FT) page placement policy. While optimizing for locality, Batch+FT relies on the GPU unified virtual memory (UVM) system to page fault data to the chiplet on which the data is first accessed. While effective for improving data locality, relying on the UVM first-touch page placement policy can introduce a substantial performance

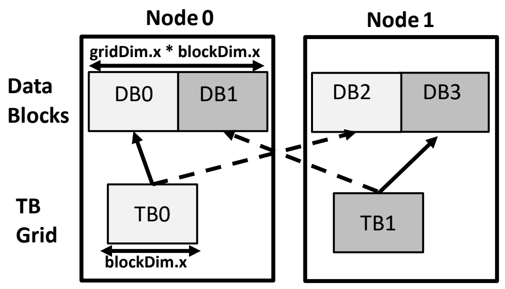


Figure 3.3. Behavior of kernel-wide partitioning in a 2-node system with 2 threadblocks that access a 4 datablock data structure with a stride of one datablock.

penalty as data must be page-faulted into GPU memory from system memory, stalling SMs for 20-50 microseconds [121]. An ideal GPU locality management system should *make an educated guess* about threadblock and data locality before execution begins, so that data and computation can *proactively* be pushed to the right location whenever possible.

The second work, by Milic et al. [10], focuses on multiple discrete GPUs. Their solution partitions both the kernel's grid and each data structure (i.e., every call to *cudaMalloc*) into N contiguous chunks, where N is the number of GPUs. Each chunk of data and threadblocks are then assigned to each respective GPU. We call this technique kernel-wide grid and data partitioning and it is pictured in Figure 3.3.

The third class of work, by Vijaykumar et al. [124] and Sun et al. [50], propose a flexible and portable software interface, called the Locality Descriptor (LD), to explicitly express data locality with a series of highly-specific APIs. Similarly, Cabezas et al. [126] rely on programmer input to shape the data placement in the program. Locality-aware threadblock clustering with code annotations was also proposed in a single GPU context [125]. Our proposed research seeks to marry the locality description benefits of these manual APIs with the transparency benefits of the locality-agnostic techniques.

Finally, the most closely related work to our own is CODA by Kim et al.[102]. CODA is a compiler-assisted index analysis framework that calculates the data accessed by each threadblock to ensure page alignment. CODA applies round-robin page and sub-page interleaving and launches static batches of threadblocks that share the same sub-page on the same node. However, CODA is only able to exploit one specific locality pattern and requires hardware changes to support sub-page address mapping.

Table 3.1 breaks down a number of common access patterns found in contemporary GPU workloads and details which prior work is able to capture them, preventing off-chip traffic. The first pattern that is *Page alignment*. If the data-placement mechanism and threadblock scheduler are unaware of how much data is accessed by a threadblock, and round-robin threadblocks among chiplets, they may not place contiguous threadblocks accessing the same page on the same chiplet. The Batch+FT scheduler launches a statically-defined batch of threadblocks (4-8 threadblocks) in a loose round-robin fashion across chiplets, in an attempt to load-balance the workload. Without knowing how big the threadblock batch should be, unnecessary off-chip accesses may occur. On the other hand, CODA is explicitly-designed to capture this pattern and to ensure that the batches are page-aligned. Kernel-wide partitioning captures this pattern as well by avoiding a round-robin scheduler and launch threadblocks in coarse-grained chunks.

The second pattern is *Threadblock-stride aware*. In this pattern, threadblocks access one chunk of data, then jump with a constant stride to read another chunk of data. Batch+FT is able to capture this pattern since the first-touch page placement policy will bring the page to the correct node. Kernel-wide partitioning and CODA are unaware of this strided behavior and will generate off-chip traffic if the stride does not accidentally match their partitioning. Figure 3.3 depicts an example of how kernel-wide partitioning works in a simple strided accesses scenario where the stride is misaligned with the system configuration, resulting in 50% off-chip accesses.

The next two patterns: *Row sharing* and *Column sharing* occur when a row or a column of threadblocks in a two-dimensional grid access the same row or column of a data structure. None of the prior techniques account for these sharing patterns, but kernel-wide partitioning is able to exploit row sharing by dividing both the grid and data structures into contiguous row-wise chunks.

The *Adjacency locality* pattern is commonly found in stencil applications where adjacent threads share data on their boundaries. The round-robin nature of Batch+FT and CODA create memory traffic at the edge of every threadblock batch. Since kernel-wide partitioning

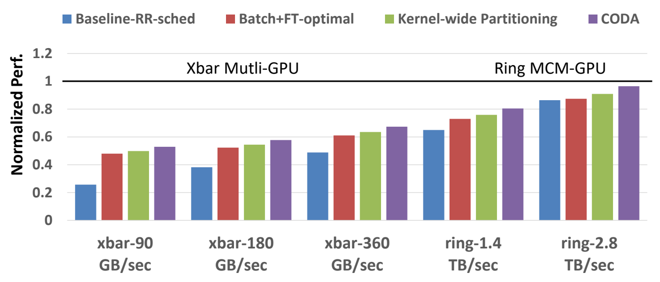


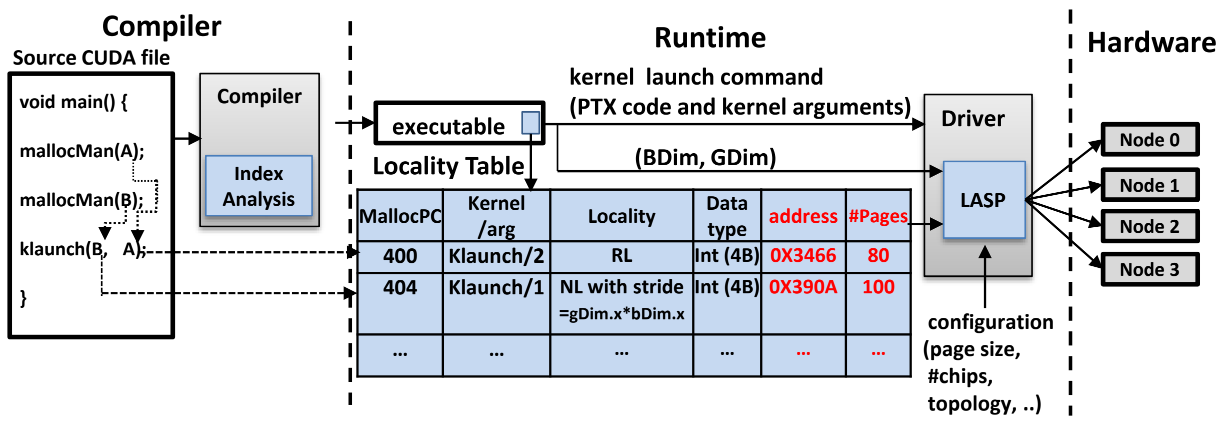
Figure 3.4. Bandwidth sensitivity analysis of state-of-the-art techniques normalized to a hypothetical monolithic GPU with the same number of SMs. Performance is averaged over the applications listed in Section 3.3.1.

is scheduled in large chunks, the number of grid cuts is minimized and so is the off-chip traffic in stencil workloads.

The *Intra-thread locality* pattern is often found in irregular workloads that have significant spatial locality in a single thread [87]. Batch+FT naturally captures this locality by moving pages to where they are first accessed. Finally, none of the existing techniques account for the size of a program's data structures and are hence input-size unaware. We explicitly design LADM to exploit all of these characteristics, which we describe in more detail in Section 3.2.

To demonstrate the relative performance of prior techniques across a variety of integration domains, we implement and evaluate several pieces of state-of-the-art work [3], [10], [102], along with a baseline round-robin placement and scheduling mechanism adopted from [5]. Figure 3.4 shows the average performance of a four GPU NUMA system with 64 SMs on each node for each evaluated technique. All values are normalized to the performance of a hypothetical monolithic GPU (where there is no NUMA access penalty to remote memories) with the same number of SMs (256).

To understand the effect topology and interconnect has on their relative performance, we simulate two different interconnection configurations connecting the four GPU nodes. First, a crossbar inter-GPU switch, similar to an NVSwitch [127], with different link bandwidths. Second, a hypothetical high-speed bi-directional ring with 1.4 and 2.8 TB/sec per-GPU to model an MCM-like topology [3]. We model optimal on-demand paging (Batch+FT-optimal) assuming page faults have zero overhead. Ideally, we would like to achieve the same monolithic chip performance with the cheapest possible interconnection. We observe that uniformly, CODA outperforms Batch+FT-optimal and kernel-wide partitioning, thanks



MallocPC	Kernel /arg	Locality	Data type	address	#Pages
400	Klaunch/2	RL	Int (4B)	0X3466	80
404	Klaunch/1	NL with stride =gDim.x*bDim.x	Int (4B)	0X390A	100
...	...	...	...	...	...

Figure 3.5. End-to-end overview of our proposed Locality-Aware Data Management System. In the locality table: MallocPC, the kernel/arg tuple, the locality type and data type are filled statically, whereas memory address and #pages are filled dynamically.

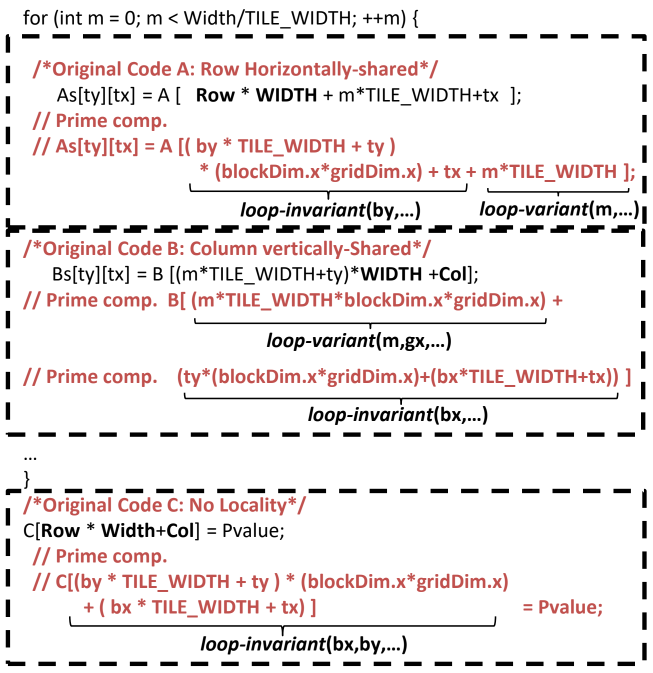
to its alignment-aware static index analysis. Yet CODA only achieves 52% and 80% of the monolithic GPU for the xbar-90 GB/sec and ring-1.4 TB/sec configurations. This implies that while CODA should be considered state-of-the-art versus other policies, there still remains significant room for improvement.

3.2 Locality-Aware Data Management

The goal of Locality-Aware Data Management is to optimize NUMA-GPU page placement, threadblock scheduling, and GPU cache management based on access patterns derived from a new threadblock-aware compiler pass with unmodified applications.

3.2.1 LADM System Design

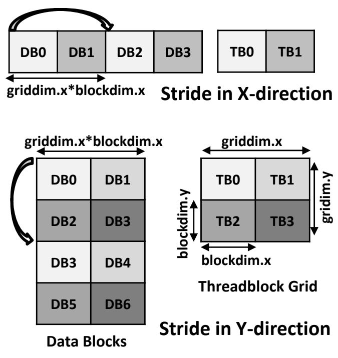
Figure 3.5 depicts an end-to-end overview of our proposed LADM mechanism. First, we perform a symbolic off-line index analysis on CUDA source code during the compilation process, detailed in Sections 3.2.2 and 3.2.3. Our analysis generates a *locality table*, which is embedded in the executable. There is one row in the table for every access to a global data pointer passed to every *__global__* CUDA function. The compiler fills the locality table with the detected locality type, data type and the MallocPC of the associated *cudaMallocManaged* call from the CPU that allocated this data structure. The MallocPC is used



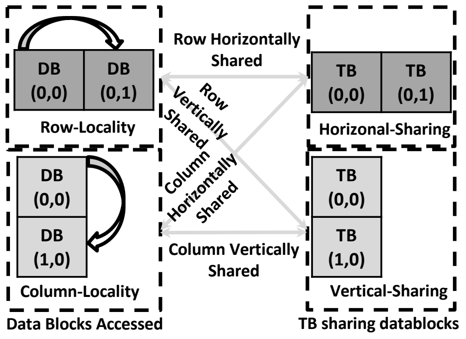
```
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
  /*Original Code A: Row Horizontally-shared*/
     As[ty][tx] = A [  Row * WIDTH + m*TILE_WIDTH+tx  ];
  // Prime comp.
  // As[ty][tx] = A [( by * TILE_WIDTH + ty )
                      * (blockDim.x*gridDim.x) + tx + m*TILE_WIDTH ];
                   loop-invariant(by,...)       loop-variant(m,...)
 /*Original Code B: Column vertically-Shared*/
    Bs[ty][tx] = B [(m*TILE_WIDTH+ty)*WIDTH +Col];
 // Prime comp.  B[ (m*TILE_WIDTH*blockDim.x*gridDim.x) +
                          loop-variant(m,gx,...)
 // Prime comp.   (ty*(blockDim.x*gridDim.x)+(bx*TILE_WIDTH+tx)) ]
                             loop-invariant(bx,...)
 ...
}
 /*Original Code C: No Locality*/
 C[Row * Width+Col] = Pvalue;
  // Prime comp.
  // C[(by * TILE_WIDTH + ty ) * (blockDim.x*gridDim.x)
       + ( bx * TILE_WIDTH + tx) ]                          = Pvalue;
                   loop-invariant(bx,by,...)
```

Figure 3.6. Matrix multiplication indices analysis

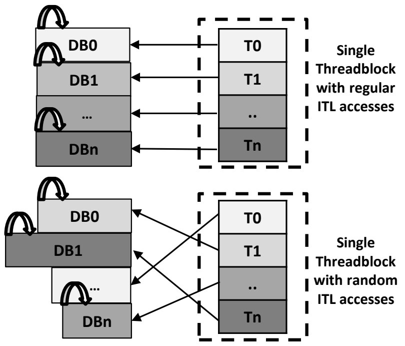
to connect the symbolic compile-time information with dynamic runtime parameters. At runtime, each *cudaMallocManaged* call inserts the number of allocated pages and address information into the kernel/argument tuples associated with this call. The mapping between *cudaMallocManaged* calls and kernel launch arguments is provided by the CPU compiler. Fortunately, the way GPU programs are written today, *cudaMallocManaged(ptr);* followed by *kernel_launch(ptr);* almost always occurs. This allows us to statically determine which *cudaMallocManaged* is associated with which kernel argument. We use traditional pointer aliasing analysis to determine the safety of this argument binding. If the static analysis is not successful, then LADM has no choice but to use a default policy for that particular call operation. Finally, on every kernel launch, the Locality-Aware Scheduling and Placement (LASP) component, described in Section 3.2.4, reads the locality table and decides the proper scheduling policy, data placement and cache strategy to reduce off-chip traffic and mitigate NUMA impact.



(a) No datablock-locality with stride in x and y direction.



(b) Row or column datablock-locality with horizontal or vertical threadblock sharing.


DB0
DB1
...
DBn
T0
T1
..
Tn
Single Threadblock with regular ITL accesses
DB0
DB1
...
DBn
T0
T1
..
Tn
Single Threadblock with random ITL accesses

(c) Intra-thread spatial locaity with regular-sized datablocks (top image) and irregular-sized random accesses (bottom image).

Figure 3.7. Common locality types found in GPU workloads. Arrows indicate threadblock motion and datablocks are shaded based on the shade of the threadblock (TB) that accesses them.

3.2.2 Threadblock-centric Locality Patterns

When work is launched to a transparent NUMA-GPU system, threads are assigned to GPUs at the threadblock granularity [10]. To create a 1:1 mapping between data placement and threadblock scheduling, we define a *datablock* as the region of data accessed by a threadblock on each iteration of the kernel's outermost loop. For example, consider the simplified kernel code for a dense $A \times B = C$ matrix-matrix multiply listed in Figure 3.6. Each thread computes one output element of the C matrix, striding through a row of A and a column of B on each loop iteration. Across an entire threadblock, each iteration of the loop will access a square region of both matrix A and B. The data accessed by the threadblock on this loop iteration is what we call a datablock. The datablock's size is directly related to the size of the data type being accessed by each thread, the dimensions of the threadblock and the components that make up the array index. Using this taxonomy, it is possible to classify the way threadblocks access data structures into one of three categories: No datablock-locality, row/column-locality, and intra-thread locality. Figure 3.7 plots a visual representation of our datablock-locality definitions.

Table 3.2. Index analysis and taken actions. *bx = blockIdx.x, by = blockIdx.y, gDimx = gridDim.x*, m is an induction variable. For the $loopInvariant$ function, if one of *bx* or *by* is not listed, then none of the terms in the equation contain that variable. For the $loopVariant$ function, if $gDimx$ is not listed, then none of the terms in the equation contain $gDimx$.

Locality Types	Index Equation	Fig	Dims	Threadblock Scheduling	Data Placement	Cache Policy
1: No datablock-locality	$loopInvariant(bx, by, ...) + stride \times m \ \forall \ stride \neq 1$	3.7a	1D/2D	Align-aware	Stride-aware	RTWICE
2: Row-locality, horizontally shared	$loopInvariant(by, ...) + loopVariant(m, ...)$	3.7b	2D	Row-binding	Row-based	RTWICE
3: Column-locality, horizontally shared	$loopInvariant(bx, ...) + loopVariant(m, ...)$	3.7b	2D	Col-binding	Row-based	RTWICE
4: Row-locality, vertically shared	$loopInvariant(by, ...) + loopVariant(m, gDimx, ...)$	3.7b	2D	Row-binding	Col-based	RTWICE
5: Column locality, vertically shared	$loopInvariant(bx, ...) + loopVariant(m, gDimx, ...)$	3.7b	2D	Col-binding	Col-based	RTWICE
6: Intra-thread locality	$loopVariant(m) = m$	3.7c	1D	Kernel-wide	Kernel-wide	RONCE
7: Unclassified	none of the above	N/A	1D/2D	Kernel-wide	Kernel-wide	RTWICE

Figure 3.7a shows the No datablock-Locality (NL) case, where threadblocks do not access the same datablocks. A simple example of an application with no datablock-locality is $C = A + B$ vector addition, where each threadblock accesses a contiguous region of A and B with no reuse or sharing. Stencil applications are another example where there is no locality among threadblocks, except among the adjacent elements. Applications that have no datablock-locality come in two forms. In the first, the kernel does not contain any loops, each datablock is computed on and then discarded. In the second, the kernel has loops and on each iteration of the loop, the threadblock strides across the data structure to another non-shared datablock. We call this movement among datablocks, *threadblock motion*. As shown in Figure 3.7a, threadblocks can access exclusive datablocks with a stride in either the x or y direction. Strided accesses frequently exist in GPGPU workloads when kernels increase the work in each thread by launching fewer threads than elements in the input data structures. Increasing work granularity per thread is a widely used optimization in CUDA programs to reduce thread initialization overhead and redundant computation [128].

It is also common for groups of threadblocks to share groups of datablocks. Figure 3.7b illustrates a sharing pattern where datablocks are accessed in either the row or column directions, by either a row or a column of threadblocks from the thread grid. For example, consider the A matrix in $A \times B = C$ matrix-matrix multiplication. A row of datablocks is shared among horizontal threadblocks. The accesses to the B matrix in matrix-matrix multiply demonstrate a different pattern. Here, a column of datablocks will be shared among vertical threadblocks. Two other possible combinations occur when rows are vertically shared and when columns are horizontally shared. We call workloads that have Row and/or Column Locality RCL workloads.

Figure 3.7c demonstrates the last type of common locality present in GPU workloads: Intra-Thread Locality (ITL). For these data structures, individual threads exhibit spatial locality across strided, regularly-sized datablocks or data-dependent, irregularly-sized datablocks. A number of prior works have shown that these applications can have significant intra-thread locality [87], [129]–[132], making shared-cache interference a significant problem.

3.2.3 Static Locality and Sharing Detection

We make the observation that static compiler analysis can make reasonable predictions about which of these three common locality patterns exist in GPU programs. We show that each locality and sharing pattern can be predicted based on an index analysis of accesses to each global data structure. The core idea is to extend traditional CPU index analysis [104] to be aware of threadblock-level definitions of parallelism. This index analysis is performed on the CUDA source code.

For regular kernels, there are two key elements we seek to determine from the static analysis: (1) the direction the threadblock moves on each loop iteration (i.e., threadblock motion), and (2) which threadblocks in the grid share the same datablocks. To determine these two variables, our source analysis begins by identifying global array accesses and expanding their index equations such that they are composed of only prime variables. We consider the following variables prime: thread IDs, block IDs, grid dims, block dims, induction variables (i.e., the loop counter) and constants. Using these variables, we then perform the analysis detailed in Algorithm 1 to classify the access, if possible.

Table 3.2 details the general index equations that are matched by our static analysis to determine which type of locality is predicted for each global array access. The compiler will attempt to match each array access to one of these 6 mutually exclusive types using Algorithm 1. The basic idea behind our index analysis is to break the index in two groups of terms. One group contains all the terms dependent on an induction variable, which we call the *loop-variant group*. The second group is composed of all the terms that are not dependent on the induction variable, which we call the *loop-invariant group*. That is, all the terms that are multiplied by an induction variable are combined in the loop-variant group, and all the remaining terms are collected in loop-invariant group. The loop-variant group determines the threadblock motion of the access, i.e., do threadblocks move horizontally or vertically through the data structure and how far do they move. Conversely, the loop-invariant terms do not change on each loop iteration and are used to determine which datablock each threadblock starts at.

To illustrate how global array-based data structures are typically accessed in GPU programs, we refer to the matrix multiplication example in Figure 3.6. The comments below the accesses to matrix A, B and C decompose the Row, Col and $WIDTH$ variables into the prime components using backward substitution and algebraic simplification. Once the access has been broken down into invariant and variant components, the compiler determines which key variables the groups are dependent on and detects the locality type using Algorithm 1.

The classification in Algorithm 1 begins by testing the special-case that the only term in the loop-variant group is the induction variable multiplied by 1. If this is the case, then we assume the access has intra-thread locality and classify the access as ITL (row 6 in Table 3.2). If that test fails, the algorithm tests if the access has no locality, by checking if the loop-invariant terms are dependent on both bx and by (for 2D threadblocks) or just bx (for 1D threadblocks). If so, we predict the access has no locality and then derive the stride by dividing the loop-variant term by m, classifying the access as row 1 of Table 3.2. The access to C in Figure 3.6 is an example of a no locality access. If neither of these first checks are true, then we search for the 4 sharing patterns in Figure 3.7b. If the loop-invariant term depends on by and not bx, then the starting datablock of all the threadblocks with the

Algorithm 1 Access classification algorithm.

```
1:  if $loopVariant(m, ...) = m$ then
2:      $access = ITL$;
3:  else if $loopInvariant(bx, by, ...)$ then
4:      access = NoLocality;
5:      stride = $loopVariant(m, ...)/m$;
6:  else if 2D Blocks then
7:      if $loopInvariant(by, ...)$ then
8:          access = ThreadblockRowShares;
9:      else if $loopInvariant(bx, ...)$ then
10:         access = ThreadblockColsShares;
11:     if $loopVariant(m, gDimx, ...)$ then
12:         access += ColumnThreadblockMotion;
13:         stride = $loopVariant(m, gDimx, ...)/m$;
14:     else if $loopVariant(m, ...)$ then
15:         access += RowThreadblockMotion;
```

same *by* (i.e., all threadblocks in the same row) will be the same. The same is true for a dependence on *bx* only, except threadblocks in the same grid column start in the same place.

After the sharing pattern is determined, the loop-variant terms are checked to determine the threadblock motion direction. If the loop-variant terms depend on $gDimx$, then we predict a whole row is being skipped on each iteration and that the threadblock motion is in the column direction, otherwise we predict that threadblocks move across a row of the data structure so long as a loop-variant term exists. Based on which combination of sharing and motion is detected, one of rows 2 through 5 in Table 3.2 is selected for accesses in 2D threadblocks. The A access in Figure 3.6 is an example of row threadblock motion, shared across threadblocks in a grid row and the B access illustrates column threadblock motion, shared across columns of threadblocks. If the array index does not match one of the locality types in Table 3.2, for example the array index contains a data-dependent component with no intra-thread locality (i.e., $X[Y[tid]]$), we leave it as unclassified (row 7 in Table 3.2) and the default placement policy is used.

After classifying each of the global array accesses in a kernel to one of the rows in Table 3.2, the compiler's work is done. The final classification of each symbol is embedded into the binary and used by the runtime system, described in the next section, to determine appropriate placement and threadblock scheduling.

3.2.4 Locality-Aware Scheduling and Page Placement

LASP is LADM's runtime system that implements page placement and threadblock scheduling based on locality patterns identified by the compiler.

LASP Data Placement

Based on the locality pattern detected for each data structure, LASP places data using the following methods.

Stride-aware placement (Row 1 in Table 3.2): To avoid off-chip traffic from strided accesses, LASP must ensure that all the datablocks accessed by a particular threadblock map to the same node. Using the stride information provided by the compiler analysis, we determine which pages need to be co-located on a given node. We interleave the pages in a round robin fashion using the page granularity given by Equation 3.1. Note that, in order to determine which threadblock maps to the next node we need to know what decision the threadblock scheduler will make. Here we assume that the aligned scheduler described in Section 3.1.2 will be used.

$$InterleavingGranularity = \left\lceil \frac{strideSize}{\#nodes} \right\rceil^{pageSize} \tag{3.1}$$

Row- and column-based placement (Rows 2-5): LASP uses row- or column-based page placement to put a whole row or column of data on the same node. For example, when rows are horizontally shared, row-based placement is used along with the row-binding scheduler (Section 3.1.2). When column-based locality is horizontally shared, column-based placement is employed with row-binding scheduler. In column-based placement, we interleave data over nodes in a round-robin fashion using Equation 3.1 where stride size is the data structure's row width.

Kernel-wide data partitioning (Rows 6 and 7): If a data structure has intra-thread locality or unclassified irregular accesses, such as graph traversal workloads. In this case, we fall back to the default data placement strategy of kernel-wide partitioning that has experimentally shown good performance for workloads that use CSR data or perform stencil operations. In these difficult to predict workloads, LADM relies on our caching mechanism described in Section 3.2.5 to further mitigate off-chip accesses by improving the L2 hit rate.

Timing of page placement and prefetching opportunities: LASP works with UVM, relieving the programmer from the burden of manually copying memory to the device. However, unlike traditional first-touch page placement, LASP makes a prediction about where every page should be placed. The pages for the data structure can be copied to the correct node as soon as the first kernel that uses a data structure is launched. We must wait until kernel launch time in order to determine the threadblock and grid sizes, which are required to compute the datablock size and strides. However, if the compiler can statically determine what the size of the first kernel launch will be, copying could potentially be started before kernel launch. It is possible that the placement derived from the first kernel launch is sub-optimal for subsequent kernel launches. Despite this potential disagreement, we find that the access pattern from the first kernel launch is often consistent with subsequent kernel launches. We leave the exploration of inter-kernel data transformations as future work.

LASP Threadblock Scheduling

Based on the locality pattern detected for each data structure, LASP schedules threadblocks using the following methods.

Alignment-aware and kernel-wide scheduler (Rows 1, 6 and 7): In the absence of any strong row or column data affinity, the scheduler attempts to load balance the work in a page-aligned fashion. To avoid the issue of page-misalignment suffered by Batch+FT [3], we can predict what the minimum threadblock batch size by using Equation 3.2, where dividing the page size by the datablock size tells us the minimum number of consecutive threadblocks ($MinTBBatch$) that should be assigned to each node to avoid misaligning datablocks and threadblocks.

$$MinTBBatch = \frac{pageSize}{datablockSize} \tag{3.2}$$

The minimum batch size will change depending on the page size and kernel arguments, since the datablock size will vary between kernels. As a result, the static batch size used in [3] will suffer when the datablocks are mis-aligned. In workloads with no locality, we have found that the datablock size is often equal to $bx \times primitiveSize$, where primitive size is 4 or 8 bytes (i.e., float versus double). Unlike CODA [102], which changes the physical page interleaving granularity and proposes fine-grained sub-page interleaving to ensure alignment, LASP keeps the page interleaving as-is and applies dynamic batch sizing using Equation 3.2 to maintain data alignment. The scheduler interleaving granularity can be any multiple of the batch size (i.e., $n \times MinTBBatch, n \geq 1$). In kernel-wide scheduling, n is the maximum possible value, in which we partition the threadblock grid into N contiguous chunks of threadblocks, where N is the number of GPU nodes.

Row- and Column-binding scheduler (Rows 2-5): The row-binding scheduler will place all threadblocks from the same row on the same node such that row-level datablock-locality is exploited. For a grid with more rows than GPU nodes, we place contiguous rows of threadblocks on each node. Similar to the row-binding scheduler, the column-binding scheduler assigns all threadblocks from the same column of the grid to the same node in order to exploit column-level datablock-locality.

Hierarchical-aware Scheduling: To exploit the fact that chiplets on the same discrete GPU will have greater bandwidth than chiplets that reside on different GPUs, the hardware and runtime system must coordinate to expose the hierarchically clustered locality domains of the underlying hardware to LASP. This allows LASP to assign adjacent threadblocks to the physically co-located chiplets on the same GPU, before moving to the next GPU. LASP employs a hierarchical affinity round-robin scheduler wherein we assign a chuck of contiguous rows or columns of threadblocks to a discrete GPU, then the assigned threadblocks are scheduled in a round-robin fashion among the chiplets within the GPU.

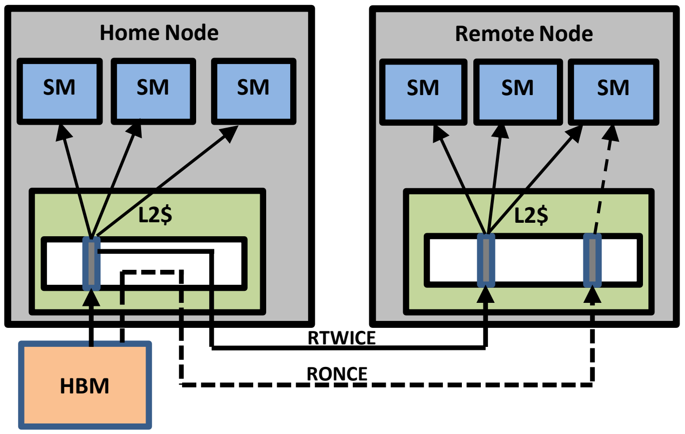


Figure 3.8. llustration of existing NUMA caching policy *cache-remote-twice* (the solid line) and our proposed *cache-remote-once* cache management strategy (the dashed line)

Data structure Locality Disagreements: Some kernels will access multiple data structures in different ways. When this happens, each structure will be placed in the way we predict is optimal, but there is only one threadblock scheduler we can select for a particular kernel. For example, in the matrix multiply example in Figure 3.6, the placement of the A matrix favors a row-binding threadblock scheduler, whereas the placement of B favors column-binding scheduling. Since it is not possible to give each data structure the scheduler that suits it best, we must pick a winner. To break the tie, we favor the scheduling policy that is associated with the larger data structure, because it will intuitively have a bigger effect on off-chip accesses, whereas smaller, frequently accessed data structures have a much greater chance of residing and hitting in the requesting node's L2. So, in our matrix multiply example, if matrix A is larger than B, we opt for a row-binding scheduling and rely on the L2 cache to reduce the off-chip traffic of the smaller matrix B. Unequal matrix sizes are commonly found in deep learning applications where a small matrix of images is multiplied by a large matrix of neuron weights.

3.2.5 Compiler-assisted Remote Request Bypassing

LASP is an efficient solution for regular workloads. However, there are additional opportunities presented in NUMA-GPU when the workloads are irregular and have intra-thread locality. Predicting the data-dependent access patterns of these irregular applications is not possible at compile time. Therefore, these irregular workloads, shown in Figure 3.7c, rely heavily on L2 caches to reduce off-chip traffic and mitigate NUMA issues [10]. We seek to improve these workloads via an intelligent cache management technique we call *cache-remote-once* that makes better use of cache in NUMA-GPUs.

Figure 3.8 illustrates the key idea of *cache-remote-once* (RONCE). In our baseline, the L2 cache is shared between local and remote traffic, similar to the dynamic shared L2 cache proposed in [10]. That is, the remote request checks the local L2 first, and if it is a miss, the request is redirected to the correct home node through the inter-chip connection. In this scenario, remote read requests are cached twice, once at the L2 cache of the home GPU and another time at the L2 cache of the GPU that sends the request. In fact, *cache-remote-twice* (RTWICE) can be beneficial in RCL workloads that count on the remote cache to minimize the NUMA effects on the victim data structure. In these workloads, remote requests are accessed by multiple SMs across the GPUs (i.e. inter-GPU locality), as shown by the solid line in Figure 3.8. However, workloads with intra-thread locality, caching requests twice is a waste of cache resources if the line is only accessed by one warp and one SM in the requesting GPU, as depicted by the dashed line in Figure 3.8. Therefore, there is no need to cache the request at the home GPU, since it may interfere with local traffic. To this end, we propose compiler-assisted remote request bypassing (CRB). In CRB, we use our compiler index analysis to determine the locality type found in the program (i.e., RCL vs ITL) and enable the RONCE bypassing policy only in ITL workloads, since our experiments show that applying RONCE for RCL may hurt the performance.

3.3 Experimental Methodology

3.3.1 Simulation Methodology

To evaluate LADM we use GPGPU-Sim version 4.0 with the recent memory system improvements from Accel-Sim simulation framework [133]. We have modified the simulator in order to model a hierarchical multi-GPU design with four GPUs connected via a switch, where each GPU is composed of four chiplets as depicted in Figure 3.1. The configuration parameters used in our system are listed in Table 3.3 and are similar to prior works [3],

[10], [134]. We have implemented the dynamically shared L2 multi-GPU cache coherence proposal from Milic et al. [10] with cache insertion policy changes that have been described in Section 3.2.5.

We have implemented the NUMA-GPU analysis proposed in the CODA system [102] and have also extended it to be aware of the GPU's hierarchical nature (H-CODA). We consider the offline profiling proposed in CODA to be an orthogonal approach to static analysis, thus we did not apply it to any evaluated technique. In all results, H-CODA is operating on top of the baseline cache coherence system. The original CODA work did not utilize any remote caching capability in hardware, but as shown in [10], utilizing remote caching in NUMA-GPUs significantly improves performance scalability on a wide range of workloads. In particular, our experiments show that enabling remote caching improves performance of general matrix multiplication (GEMM) operations by 4.8× on average, reducing off-chip traffic by 4×.

3.3.2 Workload Selection and Characterization

We run LADM on a selection of 53 scalable workloads from Rodinia 3.1 [72], CUDA SDK [73], Parboil [74], Lonestar [135] and Pannotia [136]. In addition, we include a variety of deep learning matrix math operations in which we exploit intra-layer model parallelism by running GEMM operation on multiple GPU nodes as practiced in large model training frameworks [137]. We used the optimized *sgemm* from [73], [74] as our reference implementation of GEMM and we extract layer and matrices dimensions from several popular DL networks [138]–[140]. Like prior work [3], [10], we initially pare a broader set of 53 workloads from all the benchmarks suites listed above and select only those workloads that have enough parallelism to scale-up on our simulated multi-GPU system. Of these 27 scalable benchmarks, LADM's locality detector places 24 into identifiable patterns and places 3 into the unclassified category. Table 3.4 lists the workloads used in this study, along with their detected locality types, scheduler decision, number of launched threadblocks, input size and L2 sector misses per kilo warp instructions (MPKI). It is worth noting that a workload can

Table 3.3. Multi-GPU Configuration

#GPUs	4 GPUs, 4 chiplets per GPU
#SMs	256 SMs (64 SMs per GPU, 16 SMs per chiplet)
SM configuration	Volta-like SM [133], 64 warps, 4 warp scheds, 64KB shared memory, 64KB L1 cache, 1.4 GHZ
L2 cache	16MB (1MB per GPU chiplet), 256 banks, Dynamic shared L2 with remote caching [10]
Intra-Chiplet Connect	16x16 crossbar, total BW=720 GB/s
Inter-Chiplet Connect	bi-directional ring, 720 GB/s per GPU
Inter-GPU Connect	4x4 crossbar, 180 GB/s per link, bi-directional
Monolithic Interconnect	256x256 crossbar, total BW=11.2 TB/s
Memory BW	180 GB/s per chiplet, 720 GB/s per GPU

Table 3.4. Workloads used to evaluate LADM in simulation.

Workload	Locality Type	Scheduler Decision	TB Dim	Input Size	Launched TBs	L2 MPKI
VecAdd [73]	NL	Align-aware	(128,1)	60 MB	10240	570
SRAD [72]	NL	Align-aware	(16,16)	96 MB	16384	290
HS [72]	NL	Align-aware	(16,16)	16 MB	7396	58
ScalarProd [73]	NL-Xstride	Align-aware	(256,1)	120 MB	2048	329
BLK [73]	NL-Xstride	Align-aware	(128,1)	80 MB	1920	291
Histo-final [74]	NL-Xstride	Align-aware	(512,1)	36 MB	1530	268
Reduction-k6 [73]	NL-Xstride	Align-aware	(256,1)	32 MB	2048	1056
Hotspot3D [72]	NL-Ystride	Align-aware	(64,4)	128 MB	1024	87
CONV [73]	RCL	Row-sched	(16,4)	120 MB	18432	66
Histo-main [74]	RCL	Col-sched	(16,16)	36 MB	1743	201
FWT-k2 [73]	RCL	Col-sched	(256,1)	64 MB	4096	102
SQ-GEMM [73]	RCL	Row-sched	(16,16)	128 MB	2048	61
Alexnet-FC-2 [73], [139]	RCL	Col-sched	(32,4)	400 MB	2048	8
VGGnet-FC-2 [73], [139]	RCL	Col-sched	(32,4)	76 MB	8192	8
Resnet-50-FC [73], [139]	RCL	Col-sched	(32,4)	99 MB	16384	17
LSTM-1 [73], [140]	RCL	Col-sched	(32,4)	64 MB	4096	6
LSTM-2 [73], [140]	RCL	Col-sched	(32,4)	32 MB	2048	27
TRA [73]	RCL	Row-sched	(16,16)	32 MB	16384	291
PageRank [136]	ITL	Kernel-wide	(128,1)	18 MB	23365	85
BFS-relax [135]	ITL	Kernel-wide	(256,1)	220 MB	2048	508
SSSP [136]	ITL	Kernel-wide	(64,1)	57 MB	4131	585
Random-loc [120]	ITL	Kernel-wide	(256,1)	64 MB	41013	4128
Kmeans-noTex [87]	ITL	Kernel-wide	(256,1)	60 MB	1936	158
SpMV-jds [74]	ITL	Kernel-wide	(32,1)	30 MB	4585	640
B+tree [72]	unclassified	Kernel-wide	(256,1)	16 MB	6000	112
LBM [74]	unclassified	Kernel-wide	(120,1)	370 MB	18000	784
StreamCluster [74]	unclassified	Kernel-wide	(512,1)	56 MB	1024	89

contain more than one locality type and kernel. In the table, we list the dominant locality type found in the dominant kernel.

3.3.3 Hardware Validation of LASP Principles

Like prior work, the LADM system relies on co-designed hardware and software features to maximize locality and performance. Features like remote caching, inter-GPU cache coherence, programmatically available hierarchical locality cluster information, and the capability to perform fine grained data placement among chiplets in GPUs are not present in GPUs that are available to researchers today. However, the compiler analysis provided by LASP allows us to test the software based placement of thread and data blocks on real GPUs today. We hand implemented LASP for the RCL machine learning workloads listed in Table 3.4 when running on a 4-GPU cluster within an NVIDIA DGX-1 system [109].

We use the *cudaMemAdvise* API to place the data in the correct node, assuming a 4k page. For threadblock scheduling, we used multi-kernel execution where we launch each kernel on a different GPU using CUDA streams. The kernel code was not changed and we did not employ any data replication or reactive solutions as practiced in optimized multi-GPU libraries [137], [141]. If we had access to the GPU driver, we could provide these features to the user transparently. When applying LASP's input aware scheduler and placement on real hardware, we observed 1.9× and 1.4× performance improvement compared to CODA and kernel-wide partitioning respectively. This performance improvement is achieved by preserving row- and column-locality and favoring column-binding scheduling over the row-binding scheduling when matrix B is larger than matrix A. Although this speedup required hand application coding to implement the LASP placement functionality, it is an existence proof that static analysis based locality management can lead to significant changes in performance on real systems today and into the future.

3.4 Experimental Results

3.4.1 Simulation Results of LADM

Figure 3.9 and 3.10 show the normalized performance and off-chip memory traffic for LADM, H-CODA [102] and a hypothetical monolithic GPU, when running on our simulated multi-GPU system described in Section 3.3.1. Compared to H-CODA, LADM improves the performance by 1.8× and decrease inter-GPU memory traffic by 4× on average. H-CODA and LADM are both aware of page-alignment issues. Thus, for the *VecAdd*, they both achieve the same performance. However, LADM achieves better performance in the remaining no-locality workloads due to its stride-aware placement. H-CODA fails to exploit the strided accesses found in the no-locality workloads, which causes more than 50% of

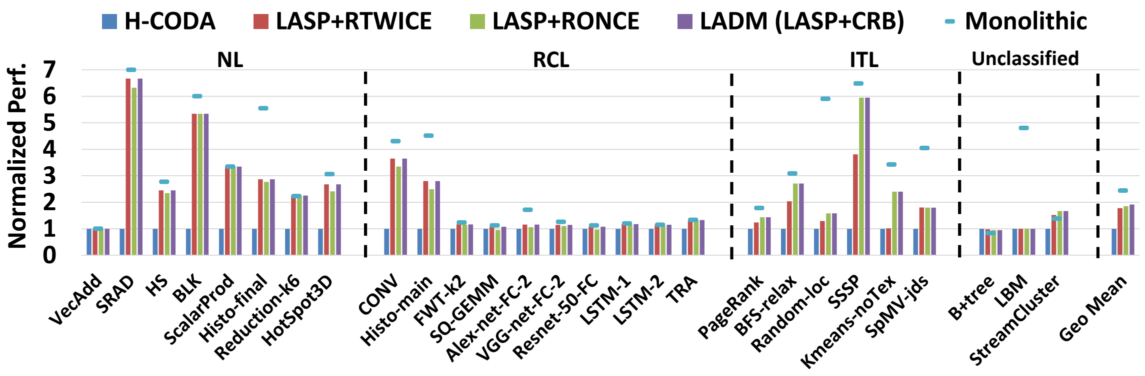


Figure 3.9. Performance of H-CODA, LASP with RTWICE and RONCE, LADM and hypothetical monolithic GPU. The data are normalized to H-CODA performance.

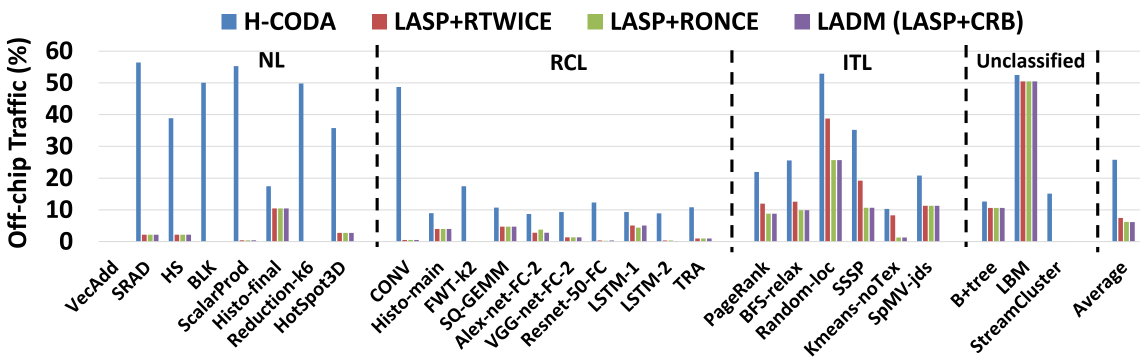


Figure 3.10. Percentage of total memory traffic that goes off-node for H-CODA vs LASP vs LADM.

memory accesses to go off-chip. Moreover, in stencil workloads, *SRAD*, *HS* and *HotSpot3D*, LADM outperforms H-CODA by 4× on average by launching contiguous threadblocks and exploiting adjacent locality of stencil workloads.

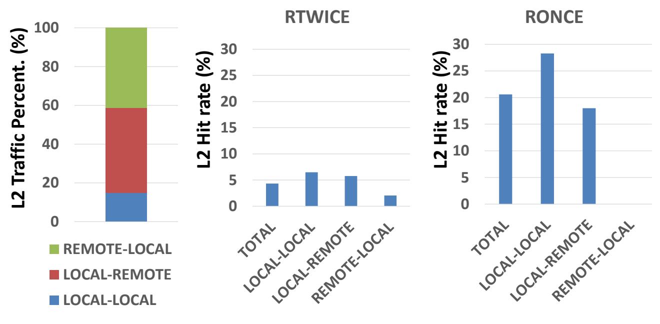
In column-locality and row-locality workloads, LADM outperforms H-CODA by 2.25×. Exploiting the column and row locality efficiently and launching the same threadblock row or column to the same chip has a substantial effect on performance. However, due to the round-robin page and threadblock interleaving of H-CODA, it fails to exploit row- and column-locality. In the machine-learning workloads, L2 remote-caching filters out off-chip traffic significantly with only 8% remaining in H-CODA. However, because of its row and column schedulers, along with its input size awareness, LADM reduces off-chip traffic further, and outperforms H-CODA by 17% on average. Although H-CODA's static analysis is agnostic to column sharing among threadblocks, it performs well when column placement is preferable. The matrix sizes in these machine-learning layers are aligned such that H-CODA's static page interleaving happens to place shared pages on the same node.

In the ITL workloads, H-CODA fails to exploit the locality between adjacent edges in graphs represented in CSR format. In contrast, LASP preserves locality by partitioning the data into large chunks of consecutive pages, improving performance by 1.7× on average. Furthermore, after applying our RONCE policy, LASP+RONCE outperforms RTWICE by an average of 38%. However, applying RTWICE outperforms RONCE by 8% on average for RCL and stencil workloads. Thus, CRB takes the best of both policies by enabling RONCE in ITL workloads and RTWICE in other locality patterns. In the unclassified workloads, LADM does not improve either performance or off-chip data accesses, except for *streamcluster*. Some workloads, like *b+tree* and *streamcluster* achieve higher performance than the monolithic GPU due to reducing bank conflicts and higher cache hit rate in the distributed L2 cache of the multi-GPU configuration. Similar trends were also observed in prior work [120].

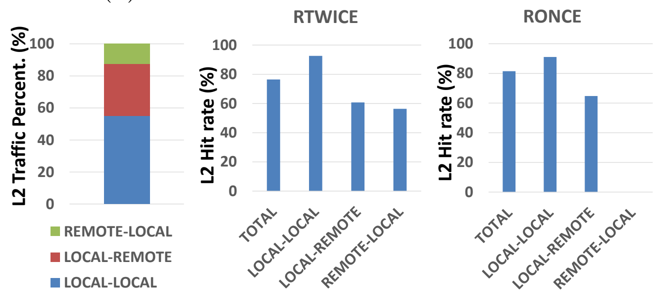
Overall, LADM outperforms H-CODA by 1.8× on average and capturing 82% of monolithic chip performance. The reasons behind the remaining 18% performance gap between LADM and monolithic chip are three-fold. First, complex indices are used, as in *lbm* and *histo*, and LADM fails to exploit their locality. Second, irregular data-dependent accesses with no intra-thread locality are frequently generated in many ITL graph workloads, and L2 remote-caching has limited impact to reduce off-chip traffic. Third, the L2 cache coherence overhead, that invalidates L2 caches between kernel boundaries, combined with global synchronization, destroys the inter-kernel locality that was exploited in the large L2 cache of the monolithic chip. Recent work [134] on hardware-supported L2 cache coherence is orthogonal to LADM and can be integrated to reduce the L2 coherence overhead.

3.4.2 Remote Request Bypassing Analysis

To better understand the remote request bypassing technique, we classify incoming L2 traffic into one of three categories: (1) LOCAL-LOCAL: A memory request generated from a



(a) *random_loc* low reuse workload.



(b) *SQ-GEMM* high reuse workload.

Figure 3.11. Case study of RONCE cache policy effectiveness on high and low reuse workloads.

local (in-node) core and serviced by local DRAM. (2) LOCAL-REMOTE: A memory request generated from a local (in-node) core. On a miss, the DRAM for the memory request is on a remote node. (3) REMOTE-LOCAL: A memory request generated from a remote node. On a miss, the DRAM for the memory request is on the local DRAM node. The total number of misses in LOCAL-REMOTE traffic is equal to the total number of REMOTE-LOCAL accesses.

Figure 3.11a presents a case study of the *random_loc* workload, where RONCE improves the performance. In *random_loc*, REMOTE-LOCAL traffic has a low hit-rate when applying RTWICE. Additionally, REMOTE-LOCAL represents 45% of the L2 traffic and causes severe contention with local accesses. Applying RONCE to bypass the REMOTE-LOCAL accesses gives more cache resources to the other traffic types and improves total L2 hit-rate by 4×. Improving the LOCAL-REMOTE hit-rate leads to fewer off-chip accesses, resulting in better performance. In contrast, Figure 3.11b plots the results when RONCE hurts the performance in *SQ-GEMM* workload. As shown in figure, REMOTE-LOCAL represents

12% of the traffic and has a relatively high hit-rate from the inter-GPU data sharing of the shared matrix. Thus, bypassing REMOTE-LOCAL leads to a performance degradation.

3.5 Summary

Recent work has shown that building GPUs with hundreds of SMs in a single monolithic chip will not be practical due to slowing growth in transistor density, low chip yields, and photoreticle limitations. To maintain performance scalability, proposals exist to aggregate discrete GPUs into a larger virtual GPU and decompose a single GPU into multiple-chip-modules with increased aggregate die area. These approaches introduce non-uniform memory access (NUMA) effects and lead to decreased performance and energy-efficiency if not managed appropriately. To overcome these effects, we propose a holistic Locality-Aware Data Management (LADM) system designed to operate on massive logical GPUs composed of multiple discrete devices, which are themselves composed of chiplets. LADM has three key components: a threadblock-centric index analysis, a runtime system that performs data placement and threadblock scheduling, and an adaptive cache insertion policy. The runtime combines information from the static analysis with topology information to proactively optimize data placement, threadblock scheduling, and remote data caching, minimizing off-chip traffic. Compared to state-of-the-art multi-GPU scheduling, LADM reduces inter-chip memory traffic by 4× and improves system performance by 1.8× on a future multi-GPU system

4. SINGLE INSTRUCTION MULTIPLE REQUEST PROCESSING FOR DATA CENTER MICROSERVICES

To meet both latency and throughput demands, contemporary data centers typically run microservices on multicore, OoO CPUs with and without Simultaneous Multithreading (SMT). Previous academic and industrial work [30], [142]–[147] has shown that current CPUs are inefficient in the data center as many on-chip resources are underutilized or ineffective. To make better use of these resource, on-chip throughput is increased [30], [144], [148] by adding more cores and raising the SMT degree [149]–[154]. Figure 4.1 visualizes the energy-efficiency and single thread latency of different processor design points, logically separated by their execution model. On the low-latency end are OoO Multiple Instruction Multiple Data (MIMD) CPUs with a low SMT-degree. Different CPU designs trade-off single thread latency for energy-efficiency by increasing the SMT-degree and moving from OoO to in-order execution. On the high-efficiency end are in-order Single Instruction Multiple Thread (SIMT) GPUs that support thousands of scalar threads per core. Fundamentally, GPU cores are designed to support workloads where single-threaded performance can be sacrificed for multi-threaded throughput. However, we argue that the energy-efficient nature of the GPU's execution model and memory system can be leveraged by low-latency OoO cores, provided the workload performs efficiently under SIMT execution. SIMT machines aggregate scalar threads into vector-like instructions for execution (i.e. a warp). To achieve high energy-efficiency, the threads aggregated into each warp must traverse similar control-flow paths, otherwise lanes in the vector units must be masked off (decreasing SIMT-efficiency) and the benefits of aggregation disappear.

We make the observation that contemporary microservices exhibit a SIMT-friendly execution pattern. Data center nodes running the same microservice across multiple requests create a natural batching opportunity for SIMT hardware, if service latencies can be met. Contemporary GPUs are ill-suited for this task, as they forego single threaded optimizations (OoO, speculative execution, etc.) in favor of excessive multithreading. Prior work on directly using GPU hardware to execute data center applications [155], [156] reports up to 6000× [156] higher latency than the CPU. Furthermore, accessing I/O resources on GPUs re-

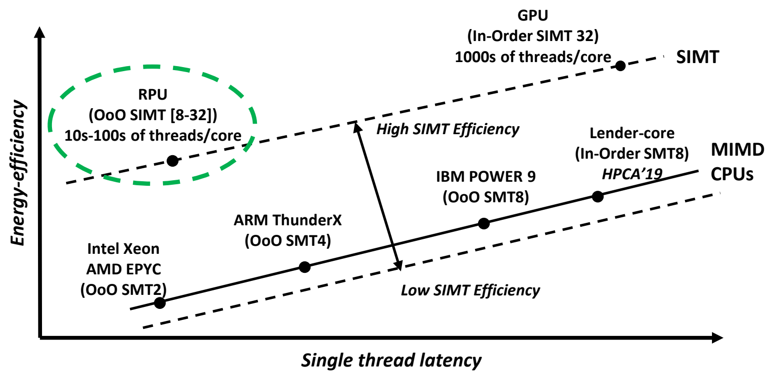


Figure 4.1. Conceptual energy-efficiency vs. single thread latency for different compute unit design points.

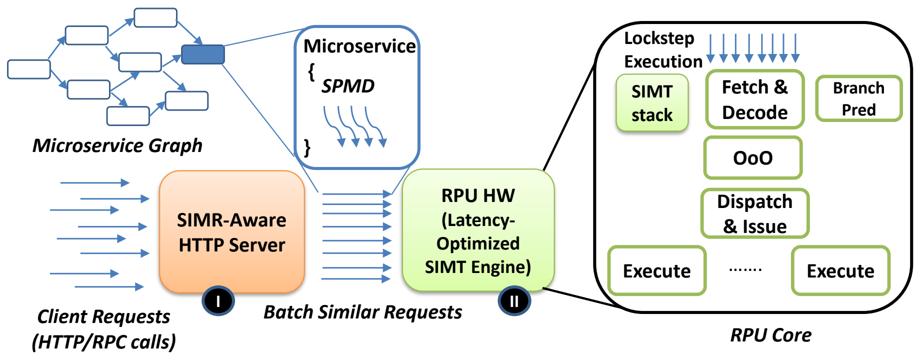


Figure 4.2. High level view of our SIMR system.

quires CPU co-ordination [156]–[160] and GPUs do not support the rich set of programming languages represented in contemporary microservices [27].

We propose replacing the CPUs in contemporary data centers with a general-purpose architecture customized for microservices: the Request Processing Unit (RPU). The RPU improves the energy-efficiency of contemporary CPUs by leveraging the frontend and memory system design of SIMT processors, while meeting the single thread latency and programmability requirements of microservices by maintaining OoO execution and support for the CPU's ISA and software stack. Under ideal SIMT-efficiency conditions, the RPU improves energy-efficiency in four ways. First, the 30% of total data center energy spent on CPU instruction supply can be reduced by the width of the SIMT unit (up to 32 in our proposal). Second, SIMT memory coalescing aggregates access among threads in the same warp, producing up to 32× fewer memory system accesses. Finally, SIMT pipelines make use of vector register files and SIMD execution units, saving area and energy versus a MIMD

Table 4.1. CPU vs RPU vs GPU Key Metrics

Metric	**CPU**	**GPU**	**RPU**
Thread/Execution Model	SMT	SIMT	SIMT
General Purpose Programming	✓	✗	✓
System Calls Support	✓	✗	✓
Service Latency	✓	✗	✓
Energy Efficiency (Requests/Joule)	✗	✓	✓

pipeline of equivalent throughput. Although the cache hit rate for SMT CPUs may be high when concurrent threads access similar code/data, bandwidth and energy demands on both cache and OoO structures will be higher than an OoO SIMT core where threads are aggregated.

Moving from a scalar MIMD pipeline to a vector-like SIMT pipeline has a latency cost. To meet timing constraints, the clock and/or pipeline depth of the SIMT execution units must be longer than that of a MIMD core with fewer threads. However, the SIMT core's memory coalescing capabilities help offset this increase in latency by reducing the bandwidth demand on the memory system, decreasing the queueing delay experienced by individual threads. In our evaluation, we faithfully model the RPU's increased pipeline latency (Section 4.3) and demonstrate that despite a pessimistic assumption that the ALU pipeline $4\times$ deeper in the RPU [161] and that L1 hit latency is $> 2\times$ higher, the average service latency is only 33% higher than a MIMD CPU chip.

Critical to the RPU's success is a well designed software system, aware of the hardware's aggregating nature that can balance SIMT efficiency and end-to-end request latency. To meet these demands, we co-design the RPU with a SIMR-aware software system pictured in Figure 4.2. The RPU executes a general-purpose CPU ISA, supporting all the same functionality as a typical CPU core, but aggregates the use of all its frontend structures over multiple threads. Table 4.1 contrasts CPUs, GPUs and the RPU at a high level. At runtime, a SIMR-aware HTTP server groups similar requests together as they enter the microservice graph. To maintain end-to-end latency requirements and keep throughput high, we introduce a similarity-aware batching technique to increase SIMT efficiency, hardware resource tuning to reduce cache and memory contention, SIMR-aware memory allocation to

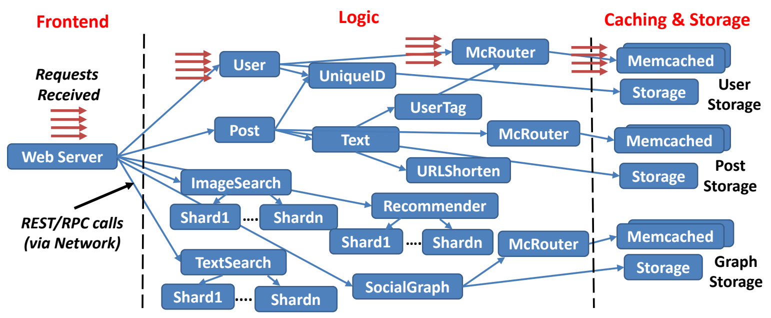


Figure 4.3. Social network microservice graph studied in this work, similar to [27].

maximize coalescing opportunities, and a system-wide batch split mechanism to minimize latency when requests traverse divergent paths with drastically different latencies.

4.1 Background and Motivation

In this section, we detail five key observations from contemporary cloud and microservices that motivate the RPU.

Key Observation #1: *Data center workloads have an abundant number of similar requests*: Public and private data centers receive a significant amount of independent requests from millions of users running the same service code [162]. These requests follow a Single Program Multiple Data (SPMD) pattern that can be efficiently leveraged on SIMT hardware [155], [156], [163].

Key Observation #2: *Microservices reduce the cache required per-thread and minimize control-flow variations between concurrent threads*: In the microservice design paradigm, a monolithic logic tier is broken down into smaller, software-friendly microservices where each is responsible for a small functionality piece of the system. Figure 4.3 depicts a simple microservice graph for a social network service similar to [27]. Each node in the data center is tasked with many threads all running the same microservice. When monolithic services are disaggregated, divergent control-flow paths are often split into different microservices. That is, *if/else* conditionals in the monolithic service are split into one service for *if* and one service for *else*. Such an organization makes it much more common that concurrent microservices on the same machine traverse exactly the same control-flow path before sending their request to the next microservice. In addition, the per-thread data cache requirement is significantly reduced, as

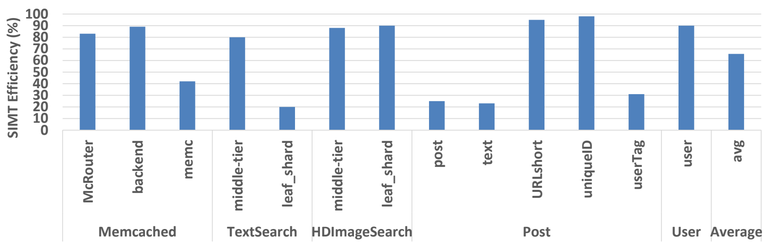


Figure 4.4. SIMT control efficiency of naive batching for some microservices.

each thread fundamentally does less work. Figure 4.4 shows the SIMT control flow efficiency of modern microservices, assuming they are batched on arrival into groups of 32 threads. On average, we are able to achieve 65% SIMT efficiency when applying naive batching. In section 4.2.2, we propose optimized batching techniques, which bring efficiency to 91%.

Key Observation #3*: Modern data centers already rely on request batching*: In order to enable SIMT execution, requests have to be batched and executed together. Batching can introduce additional latency to wait and group a complete batch of requests. However, batching is already heavily used in data centers and employed in at least one microservice on each network path, for example: (1) Deep Learning inference batches requests to increase accelerator compute throughput [22], (2) key-value store applications, like memcached [164], batch to amortize the network overhead, (3) streaming graph analytics [165] batch to alleviate lock contention, and (4) dynamic power management [166]–[169] applies batching to save power. Therefore, if we apply batching to exploit request similarity, the batching overhead is amortized, as there are already individual microservices on the same path that employ batching. Instead of enabling batching on specific microservices, we propose to implement batching to all other microservices from end-to-end scenarios, as depicted in Figure 4.3.

Key Observation #4*: In the data center, all throughput gains must be made under a tight latency constraint*: The trade-off between brawny and wimpy cores in the data center has been a well-studied problem [170], [171]. However, the use of wimpy cores has not been widely adopted by data center providers [170]. Although, under the same power budget, wimpy cores can increase throughput [172], they have higher task execution latency than brawny cores, increasing total request latency and making them ill-suited for the data center's QoS-driven workloads unless their single-thread latency is no worse than 2× that of brawny

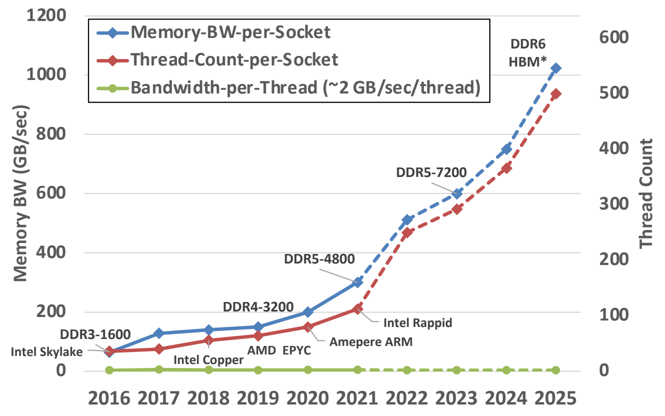


Figure 4.5. Off-chip DRAM BW and Thread per socket scaling.

cores [170], [172]. The same argument applies for GPUs, that have high energy-efficiency, but have unacceptably high service latency, 6000× worse than CPUs for SPEC-Web [156], and 10× worse for memcached [155].

Key Observation #5: *Future data center nodes need to increase their on-chip thread count*: Previous academic and industrial work [30], [142]–[147] has shown that current CPUs are inefficient when executing data center workloads as there are many underutilized resources. They suggest that an increase in the number of threads on-chip is necessary to better use these resources [30], [143], [144]. Figure 4.5 depicts the off-chip bandwidth and thread count per socket scaling in the future. CPU vendors typically ensure 2 GB/sec of DRAM BW per thread. If this is the case, we need to provide up to 256 threads per socket with DDR5 [173]–[176] and 512 threads with DDR6 [177] and HBM [178] to utilize the available off-chip BW. The industry standard to increase on-chip throughput is by adding more chiplets [149], [150], cores [151], [152] and increasing the SMT degree [153], [154]; however we argue that introducing SIMT to OoO CPU cores will provide a more energy-efficient mechanism to scale on-chip throughput.

Given these five observations, we design our RPU hardware and software system to exploit the similarity among requests in microservices through intelligent batching. The RPU's OoO SIMT frontend is able to meet the latency constraints of contemporary services, while improving upon the energy-efficiency and thread-density of modern CPUs. In the next section, we discuss our system's design.

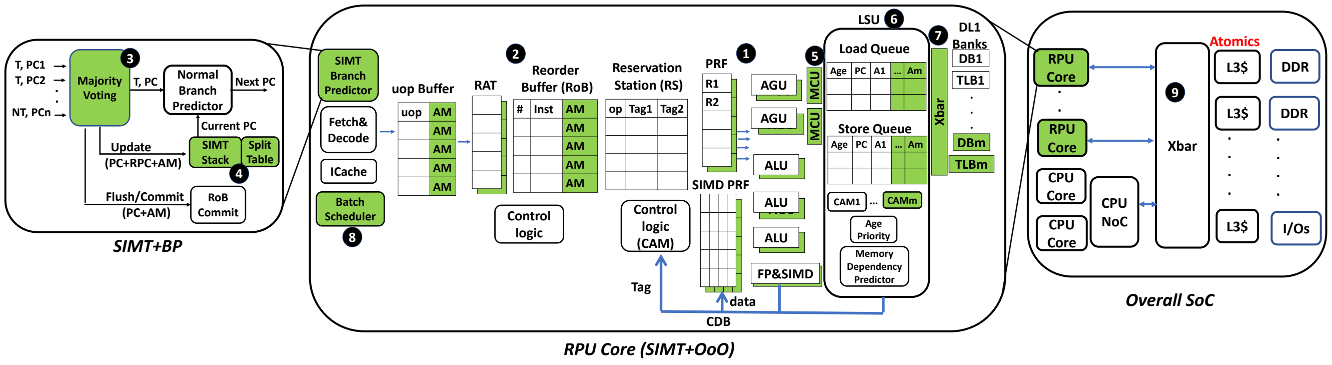


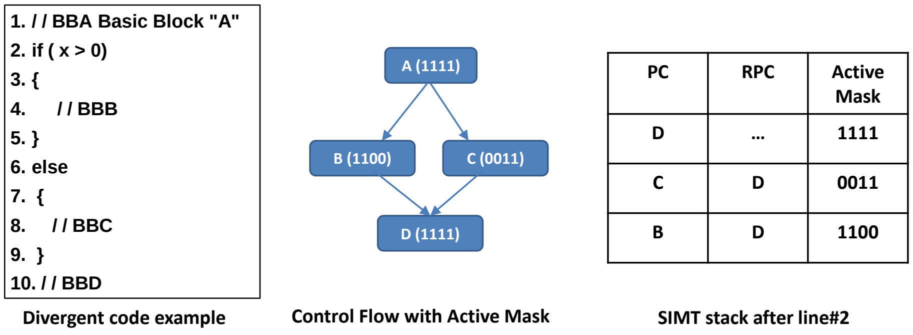
Figure 4.6. RPU hardware overview. Changes to the OoO core needed to support SIMR execution are highlighted in green.

4.2 SIMR System

Figure 4.2 presents a high level overview of our SIMR system. Groups of independent Remote Procedure Call (RPC) or HTTP requests are received by our SIMR-Aware server. The server (I in Figure 4.2) groups requests into a batch based on each request's Application Program Interface (API) similarity and argument size. The batches in the RPU are analogous to warps in a GPU. Our batch size is tunable based on resource contention, desired QoS, arrival rate and system configuration (Section 4.2.2 explores these parameters). Then, the server launches a service request to the RPU driver and hardware. The RPU hardware (II) executes the batch in lock-step fashion over the OoO SIMT pipeline (Section 4.2.1).

4.2.1 RPU Hardware

Figure 4.6 presents a detailed overview of our RPU hardware. Each RPU core is similar to a brawny OoO CPU core, except hardware is added to perform multithreading in a SIMT fashion. The design philosophy of the RPU is that the area/power savings gained by SIMT execution and amortizing front-end (e.g., OoO control logic, branch predictor, fetech&decode) are used to increase the thread context and throughput at the back-end (1 in Figure 4.6), e.g., register file, execution, and cache resources; thus we still maintain the same area/power budget and improve overall throughput/watt. Our RPU chip contains multiple RPU cores, and a few CPU cores. The role of the CPU cores is to run the OS process, HTTP server, and RPU driver.



```
1. / / BBA Basic Block "A"
2. if ( x > 0)
3. {
4.     / / BBB
5. }
6. else
7.  {
8.     / / BBC
9.  }
10. / / BBD
```

Divergent code example

Control Flow with Active Mask

PC	RPC	Active Mask
D	...	1111
C	D	0011
B	D	1100

SIMT stack after line#2

Figure 4.7. IPDOM analysis with HW SIMT stack.

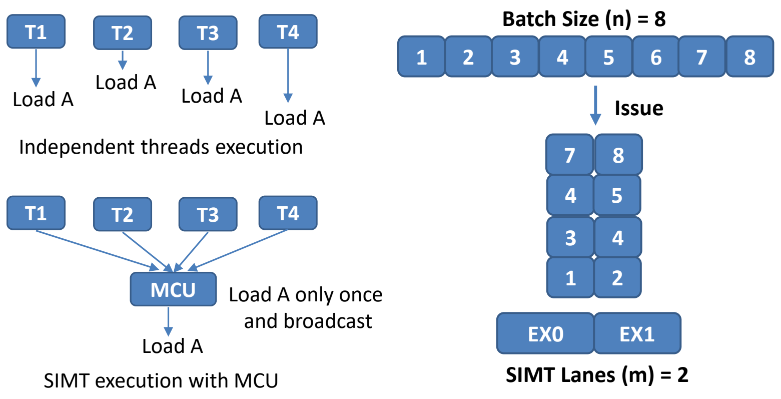
OoO SIMT Pipeline: When merging the RPU's SIMT pipeline with speculative, OoO execution, we assume the following design principles. First, the active mask is propagated with the instruction throughout the entire pipeline (❷). Therefore, instruction buffer, reorder buffer, and register alias table (RAT) entries are extended to include the active mask (AM). Second, to handle register renaming of the same variable used in different branches, a micro-op is inserted to merge registers from the different paths [179]. Third, the branch predictor operates at the batch (or warp) granularity, i.e., only one prediction is generated for all the threads in a batch. When updating the branch history, we apply a majority voting policy of branch results (❸). For mispredicted threads, their instructions are flushed at the commit stage and the SIMT stack is updated accordingly. Adding the majority voting circuitry before the branch prediction increases the branch execution latency and power. We account for these overheads in our evaluation, detailed in Section 4.3.

Control Flow Divergence Handling: To address control flow divergence, a hardware SIMT stack (❹) is employed to serialize divergent paths [180], [181]. Upon receiving a new request from the server, the driver performs a just-in-time Immediate Post-Dominator (IPDOM) analysis on the CPU binary [182], and send this reconvergence information to the hardware along with the program. Note that this analysis needs only be done once per application binary. Figure 4.7 illustrates a simple example of IPDOM analysis and how the SIMT stack interacts with divergent control flow. This mechanism is identical to the stack used in contemporary GPUs. When batched requests execute divergent control flows, the paths are serialized, and each path is associated with a corresponding active mask and reconvergence PC. The serialization overhead is minimized by intelligent batching techniques that minimize control flow divergence, which we describe in Section 4.2.2.

Running threads in lock-step execution with IPDOM reconvergence can induce deadlock when programs employ inter-thread synchronization[183]–[185]. There have been several proposals to alleviate the SIMT-induced deadlock issue on GPUs. Fundamentally, all the proposed solutions rely on multi-path execution to allow control flow paths not at the top of the SIMT stack to make forward progress. In the RPU, we use delayed reconvergence with warp split table (❹) introduced by ElTanawy and Aamodt [183], which requires an additional 1KB of storage/batch (accounted for in Section 4.3). NVIDIA's independent thread scheduling [185], introduced in Volta, is a SIMT stackless solution that potentially requires programmer input to optimize SIMT efficiency. We leave the exploration of a transparent stackless solution as future work.

Memory Coalescing: To improve memory efficiency, a low-latency memory coalescing unit (MCU) is placed before the load and store queues (❺). As described in Figure 4.8a, the MCU is designed to coalesce memory accesses to the same cache line from threads in a single batch, making better use of cache throughput and avoiding cache access serialization. The MCU filters out accesses to shared inter-request data structures that might exist in the heap or data segments [144]. To balance the need for a low cache hit latency and avoiding divergent accesses serialization, the MCU only detects the two most common memory coalescing scenarios: when all threads access the same word, or when threads access consecutive words from the same cache line. This is unlike the complex sub-batch sharing in GPU data coalescing [180], [186] that increases memory access latency to detect more complex locality patterns [187].

LD/ST Unit: In our MCU, if neither simple pattern is detected, the number of accesses generated will equal the number of active SIMT lanes. All accesses from the same instruction will allocate one row in the load or store queue (❻), sharing the same PC and age fields/logic, and thus amortizing the memory scheduling and dependence prediction [188] overhead. The entries of the RPU's LD/ST queues are expanded such that each row can contain as many addresses as there are SIMT lanes. This expansion is accounted for in Section 4.3.



(a) MCU (b) Sub-batch interleaving

Figure 4.8. MCU and sub-batch interleaving to improve memory efficiency and hiding front-end latency respectively

Cache and TLB: To serve the throughput needs of many threads, while achieving scalable area and energy consumption, the RPU uses a banked L1 cache. The load/store queues are connected to the L1 cache banks via a crossbar (❼). To ensure TLB throughput can match the L1 throughput, each L1 data bank is associated with a TLB bank. Since the interleaving of data over cache banks is at a smaller granularity than the page size, TLB entries may be duplicated over multiple banks. This duplication overhead reduces the effective capacity of the DTLBs, but allows for high throughput translation on cache+TLB hits. As a result of the duplication, all TLB banks are checked on the per-entry TLB invalidation instructions [189]. Sections 4.2.2 and 4.2.2 discuss how we alleviate contention to preserve intra-thread locality and achieve acceptable latency via batch size tuning.

Sub-batch Interleaving: Previous work [143], [144] show that data center workloads tend to exhibit low IPC per thread (a range of 0.5-2, the average is 1 out of 5), due to long memory latencies at the back-end and instruction fetch misses at the front-end [30], [145]. To increase our execution unit utilization, we implement sub-batch interleaving [190], [191] as depicted in Figure 4.8b. By decreasing the number of SIMT lanes per execution unit, we issue threads over multiple cycles. Sub-batch interleaving along with OoO scheduling can hide nanosecond-scale latencies, increasing our IPC. Another advantage of sub-batch interleaving is that we can skip issue slots of non-active threads to mitigate control divergence penalty and support smaller batches of execution [190]. To hide longer microsecond-scale

Table 4.2. CPU vs RPU vs GPU Architecture Differences

Metric	**CPU**	**GPU**	**RPU**
Core model	**OoO**	In-Order	**OoO**
Freq	**High**	Moderate	**High**
ISA	**ARM/x86**	HSAIL/PTX	**ARM/x86**
Programming	**General-Purpose**	CUDA/OpenCL	**General-Purpose**
Thread grain	**Coarse grain**	Fine grain	**Coarse grain**
TLP per core	Low (1-8)	Massive (2K)	Moderate (8-32)
Thread model	SMT	**SIMT**	**SIMT**
Consistency	Variant	**Weak+NMCA**	**Weak+NMCA**
Coherence	Complex	**Relaxed Simple**	**Relaxed Simple**
Interconnect	Mesh	**Crossbar**	**Crossbar**

latencies [33], multiple batches are interleaved via hardware batch scheduling (❽) in a coarse-grain, round-robin manner with zero-overhead context switching.

Weak Consistency Model: To exploit the fact that requests rarely communicate and exhibit low coherence, read-write sharing or locking [143], [144], as well as extensive use of eventual consistency in data center [192], we design the memory system to be similar to a GPU, i.e., weak memory consistency with non-multi-copy-atomicity (NMCA) and a simple, relaxed coherence protocol with no-transient states or invalidation acknowledgements, similar to the ones proposed in HMG [134] and QuickRelease [193] [1]. That is, cache coherence and memory ordering are only guaranteed at synchronization points (i.e., barriers, fences, acquire/release), and all atomic operations are moved to the shared L3 cache. Therefore, we no longer have core-to-core coherence communication, and thus we replace the commonly-used mesh network in CPUs with a higher-bisection-bandwidth, lower-latency core-to-memory crossbar (❾).

CPU vs GPU vs RPU

Table 4.2 lists the key architectural differences between CPUs, GPUs and our RPU. The RPU takes advantage of the latency-optimizations and programmability of the CPU while exploiting the SIMT efficiency and memory model scalability of the GPU. Finally, Table 4.3

[1]↑In fact, some CPU ISA, like ARM [194], [195] and POWER [196], already supports weak consistency model with non-multi-copy-atomicity in their specifications [197].

Table 4.3. CPU inefficiencies in the data center

Data center characteristics & CPU inefficiency	**RPU's mitigation**
Request similarity [156] & high frontend power consumption [12]	SIMT execution to amortize frontend overhead
Inter-request data sharing [144]	Memory coalescing and an increase in the number of threads sharing private caches
Low coherence/locks [143], [144] and eventual consistency [192]	Weak memory ordering, relaxed coherence with non-memory-copy-atomicity & higher bandwidth core-to-memory interconnect
Low IPC due to frequent frontend stalls and memory latency [30], [33], [142]–[145]	Multi-thread interleaving
DRAM & L3 BW are underutilized, data prefetchers are ineffective [31], [143], [144], [146]	High thread level parallelism (TLP) to fully utilize BW
Microservice/nanoservice have a smaller cache footprint [27]	High TLP and decrease L1&L2 cache capacity/thread

summarizes a set of data center characteristics that create inefficiencies in CPU designs and how the RPU improves them.

An Examination of SMT vs SIMT

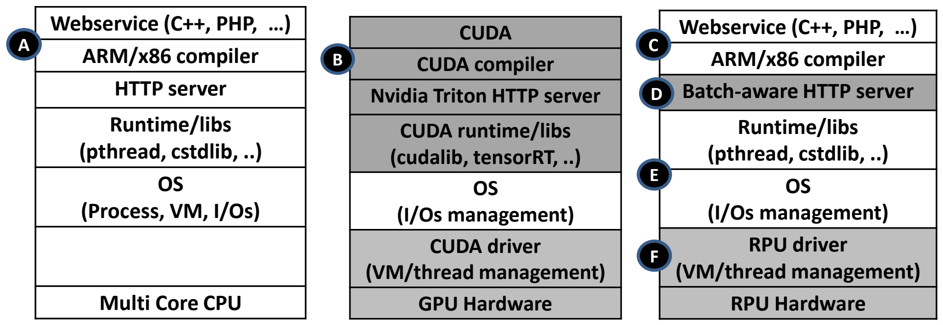
This subsection examines why the RPU's SIMT execution is able to outperform MIMD SMT hardware for data center workloads. Equation 4.1 presents an analytical computation of the RPU's energy efficiency (EE) gain over the CPU. In Equation 4.1, n is the RPU batch size, *eff* is average RPU SIMT efficiency, and r is the ratio of memory requests that exhibit inter-thread locality within a single SIMR batch. CPU energy is divided into frontend OoO overhead (i.e., fetch, decode, branch prediction and OoO control logic), execution (including, register reading and instruction execution), memory system (including, private caches, interconnection and L3 cache), and static energies.

$$EnergyEfficiency = \frac{CPU_{Energy}}{RPU_{Energy}} = \frac{\frac{Exec_{Energy} + Mem_{Energy} +}{Exec_{Energy} + (1-r)Mem_{Energy} +}\;Front\_OoO_{Energy} + Static_{Energy}}{\frac{1}{n*eff}[r*Mem_{Energy} + Front\_OoO_{Energy} + Static_{Energy}] + SIMT_{Overhead}} \tag{4.1}$$

In Equation 4.1, we make the conservative assumption that the RPU's energy consumption in back-end instruction execution and non-coalesced memory accesses are no different than a MIMD CPU. However, front-end and OoO overheads are amortized in the RPU by running threads in lock step; hence the energy consumed for instruction fetch, decode, branch prediction, CAM Tag accesses [198] for register renaming, reservation station, and store-to-load forwarding are all consumed only once for all the threads in a single batch (see Figure 4.6). In scalar CPU designs, the front-end and OoO overheads have to be consumed for each thread. Even with SMT, the entire CPU pipeline is partitioned among the simultaneous threads. Threads on the same core are executed independently [153], [154], [199], which fails to exploit thread similarity and increases single thread latency.

Coalesced memory accesses are also amortized in the RPU by generating and sending only one access for a batch to the memory system. While private cache hits and MSHR merges can filter out some of these coalesced accesses in a SMT design, you have to guarantee that the simultaneous threads are launched and progress together to capture this inter-thread data locality [200], [201] and you still pay the energy cost of multiple cache accesses. Furthermore, since SIMT can execute more threads/core given the same area constrains, the reach of its locality optimizations is wider.

The final metric SIMT execution amortizes is static energy. The RPU improves throughput/area and has a smaller SRAM budget/thread compared to an SMT core. In summary, if these amortized components consume 50-80% of the total CPU energy [12], an anticipated 2-5x energy efficiency gain can be achieved with the RPU if SIMT efficiency is high and accesses are frequently coalesced. It is worth mentioning that the RPU introduces an energy overhead for SIMT stack updates, active mask propagation, MCUs and multi-bank L1/L2 cache arbitration. However, at high SIMT efficiency, the energy savings from the amortized metrics greatly outweigh the SIMT management overhead. In the next section, we experimentally show how much SIMT control and memory efficiency exist in microservices workloads and explore the effect of different batch sizes.


Webservice (C++, PHP, ...)
A
ARM/x86 compiler
HTTP server
Runtime/libs (pthread, cstdlib, ..)
OS (Process, VM, I/Os)
Multi Core CPU
CUDA
B
CUDA compiler
Nvidia Triton HTTP server
CUDA runtime/libs (cudalib, tensorRT, ..)
OS (I/Os management)
CUDA driver (VM/thread management)
GPU Hardware
Webservice (C++, PHP, ...)
C
ARM/x86 compiler
D
Batch-aware HTTP server
Runtime/libs (pthread, cstdlib, ..)
E
OS (I/Os management)
F
RPU driver (VM/thread management)
RPU Hardware

(a) CPU SW Stack (b) GPU SW Stack (c) RPU SW Stack

Figure 4.9. Hardware/Software Stack of CPU vs GPU vs RPU for microservices programming

4.2.2 SIMR Software Stack

Figure 4.9 compares the RPU's software (SW) stack, to that of the CPU and GPU. GPU computing (Ⓑ in Figure 4.9) generally requires the programmer to use a specialized language, like CUDA, and (in the case of NVIDIA) uses a closed-source compiler, runtime, driver, and ISA. These all restrict programmer productivity. While GPUs have been successful for accelerating the DL inference, they are poorly suited for others with middling parallelism and tight deadlines.

Microservice developers typically use a variety of high-level, open-source programming languages and libraries (Ⓐ). For the RPU, we maintain the traditional CPU software stack (Ⓒ, Ⓔ), changing only the HTTP server, driver and memory management software. The RPU is ISA-compatible with a traditional CPU.

The role of our HTTP server (Ⓓ) is to assign a new software thread to each incoming request [202], [203]. The SIMR-aware server groups requests in a batch based on each request's Application Program Interface (API) similarity and argument size (see Section 4.2.2), then sends a launch command for the batch to the RPU driver with pointers to the thread contexts of these requests.

The RPU driver (Ⓕ) is responsible for runtime batch scheduling and virtual memory management. The driver overrides some of the OS system calls related to thread scheduling, context switching, and memory management, optimizing them for batched RPU execution. For example, context switching has to be done at the batch granularity (Section 4.2.2), and

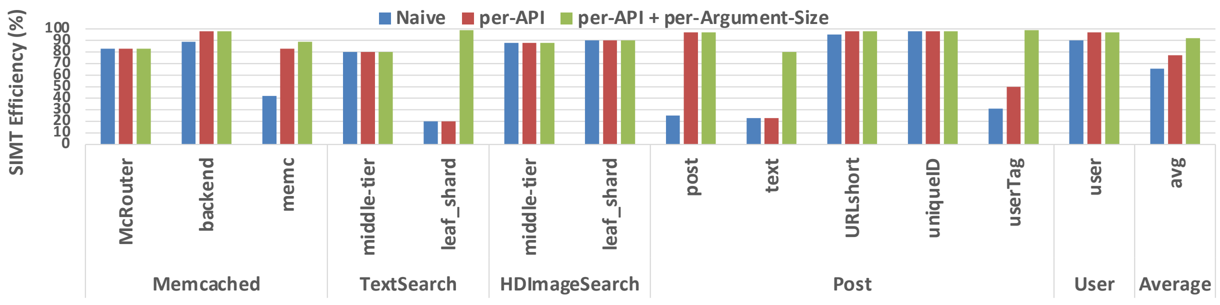


Figure 4.10. SIMT control flow efficiency with different request batching policies (Batch Size = 32)

memory management is optimized to improve memory coalescing opportunities at runtime (Section 4.2.2).

To ensure efficient SIMT execution, the software stack's primary goals are to: (1) minimize *control flow divergence* by predicting and batching requests control flow (Section 4.2.2), (2) reduce *memory divergence* and alleviate cache/memory contention (Sections 4.2.2, 4.2.2, 4.2.2) with batch tuning and SIMR-aware virtual memory mapping, and (3) alleviate *network/storage divergence* through system-wide batch splitting (Section 4.2.2).

SIMR-Aware Batching Serve

A key aspect to achieve high energy efficiency is to ensure batched threads follow the same control flow to minimize control divergence. To achieve this, we need to group requests that have similar characteristics. Thus, we employ two heuristic-based proof-of-concept batching techniques. First, we group requests based on API or RPC calls. Some microservices may provide more than one API, for example, *memcached* has *set* and *get* APIs, *post* provides *newPost* and *getPostByUser* calls. Therefore, we batch requests that call the same procedure to ensure they will execute the same source code. Second, we group requests that have similar argument/query length. For example, when calling the *Search* microservice, requests that have long search query (i.e., more words) are grouped together as they will probably have more work to do than the smaller ones. Figure 4.10 shows the SIMT efficiency (i.e., = #scalar-instructions / (#batch-instructions $\times$ batch-size)) for naive batching (based on arrival time) and an optimized per-API and per-argument batching. We assume a batch size of 32 requests for all microservices and we calculate the average over 75 batches (2400 requests). As shown in Figure 4.10, batching per-API improves SIMT efficiency for many microservices, up to 2x improvement is noticed in *memcached*, and 4x in *Post* microservices. When taking into account per-argument length batching, the overall SIMT efficiency is further improved by 20% on average and up to 5x better on the *TextSearch-leaf* and *post-text* microservices. In total, we are able to achieve 91% SIMT efficiency over 13 microservices.

It is worth mentioning that we achieve this SIMT efficiency while making the following assumptions. First, some of these microservices are not well optimized and employ coarse-grain locking which affects our control efficiency negatively due to critical section serialization and lock spinning. In practice, optimized data center workloads rely on fine-grain locking to ensure strong performance scaling on multi-core CPUs [144], [203]. In our experiments, if threads access different memory regions within a data structure we assume that fine-grained locks are used for synchronization. We also assume that a high-throughput, concurrent memory manager is used for heap segment allocation [204]–[206] rather than the C++ glibc allocator that uses a single shared mutex. Finally, the microservice *HDSearch-midtier* applies kd-tree traversal and contains data-dependent control flow in which one side of a branch contains much more expensive code than the others. To improve SIMT efficiency in such scenarios, we make use of speculative reconvergence [207] to place the IPDOM synchronization point at the beginning of the expensive branch.

Stack Segment Coalescing

Similar to the local memory space in GPUs [180], [208], Figure 4.11 depicts how the RPU driver and TLB hardware allocate and map stack memory from different threads in the same batch to minimize memory divergence. The interleaving is static and transparent to the compiler and the programmer. When the runtime system calls *mmap* to allocate a new stack segment for a thread [209], [210], we ensure that the stack segments for all the threads in a batch are contiguous (ⓐ in Figure 4.11). In hardware, we detect accesses to stack addresses and apply an interleaved data mapping (ⓑ), such that stack segments from different threads are interleaved every 4 bytes in the physical address space (ⓒ). The RPU's address generation unit overrides the stack base of all active threads with the stack

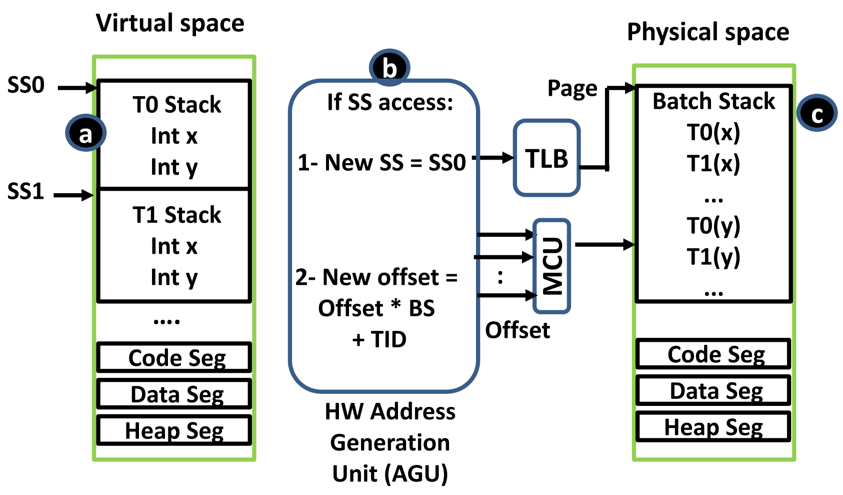


Figure 4.11. Stack Segment (SS) coalescing (physical stack page size = virtual page size * batch size) with 4-byte interleaving.

base of thread 0, thus we only need one TLB translation per stack access. A hardware offset mapping uses the thread ID (TID) of the accessing thread as an index into the S0 space to determine where the value resides in physical memory. This hard mapping prevents threads from accessing other thread's stack data, which is allowed in CPU programming. To alleviate this issue, we calculate the target stack segment TID of each access based on the access' virtual segment address, i.e. $TargetTID = (SS - SS0)/StackSize$, exploiting the fact that stacks are allocated consecutively in the virtual space. If the accessing thread has permission to access the target thread's stack (discussed further in Section 4.5), then the TargetTID is used, allowing inter-thread stack accesses. It is worth noting that GPU programming languages avoid this issue by making stack values thread-loca.

Figure 4.12 demonstrates the effectiveness of our stack interleaving and heap memory coalescing policies (previously described in Section 4.2.1). Figure 4.12 plots the total number of L1 accesses in the RPU, normalized to a MIMD CPU, when both are executing 640 threads. The RPU's 32-thread batches generate on average 3x less accesses than the CPU. The causes of this traffic reduction are two-fold. First, many of our middle tier microservices contain significant stack segment accesses (up to 90% in the Post microservices) caused by frequent procedure/system calls, push/pop argument passing, and reading/writing local variables. Our stack segment interleaving technique coalesces all these accesses and generates less traffic compared to the CPU. For example, pushing an 8-byte address in each thread of

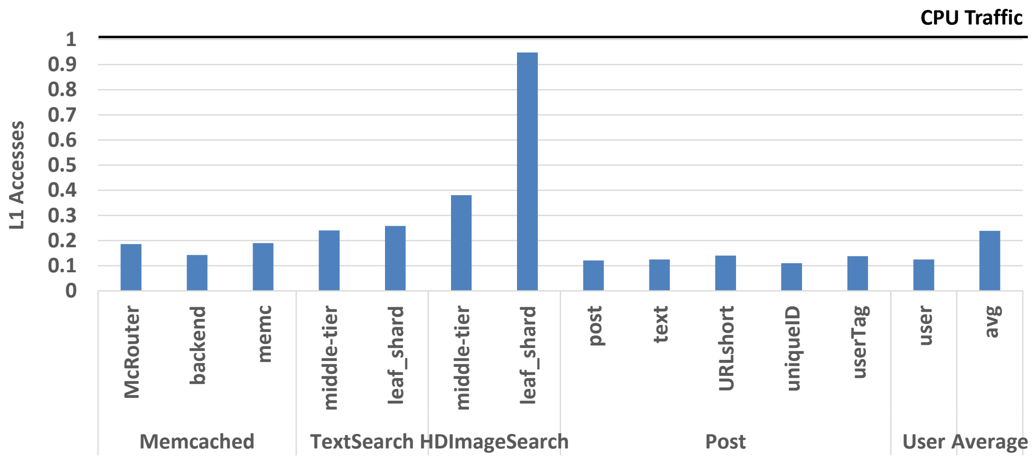


Figure 4.12. RPU L1 accesses, normalized to CPU accesses

a 32-thread batch onto the stack generates 8 accesses (8B x 32 threads / 32B cache lines); however, in the CPU, 32 accesses are generated. Second, microservices typically share some global data structures and constant values in the heap and data segments [144] respectively. In the RPU, accesses to this shared data are coalesced within the MCU and loaded once for all the threads in a batch, improving L1 data locality. While traffic reduction is significant in many cases, back-end data-intensive microservices, like *HDSearch*, still exhibit high traffic as each thread contains private data structures in the heap with little sharing, resulting in frequent divergent heap accesses.

Batch Size Tuning and Memory Contention

Previous work [27] shows that micro and nanoservices typically exhibit a low cache footprint per thread, as services are broken down into small procedures and read-after-write inter-procedure locality is often transferred to the system network via RPC calls. To exploit this fact, we increase the number of threads per RPU core compared to traditional CPUs. Figure 4.13 shows the L1 MPKI of a single threaded CPU with 64KB of L1 cache and an RPU with different batch sizes (32, 16, 8, 4) and 256KB of L1 cache. Interestingly, many of our microservices can run at a batch size of 32 threads and require only 8KB/thread without thrashing the L1 cache. More importantly, for these microservices, the L1 MPKI is significantly improved compared to the CPU. This is because memory coalescing reduces the

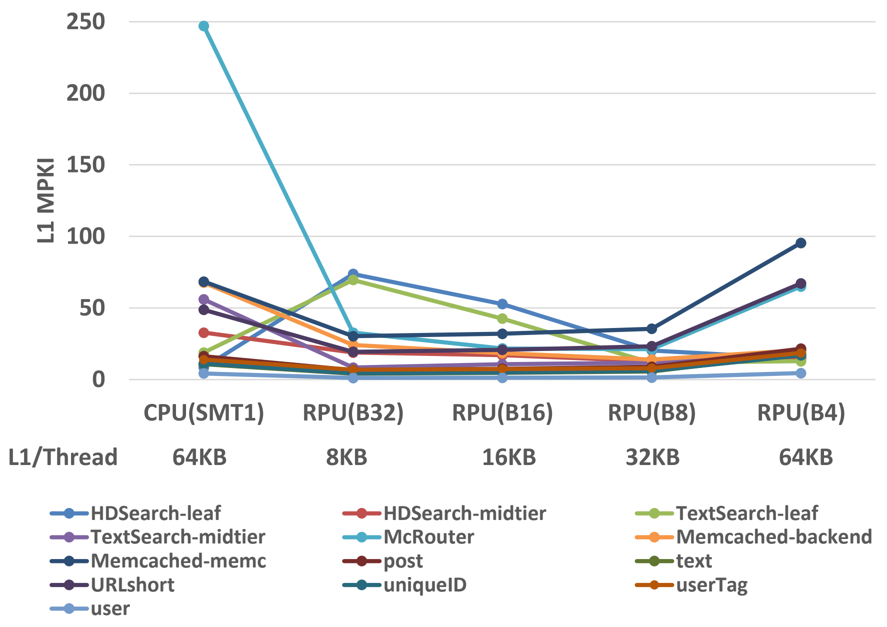
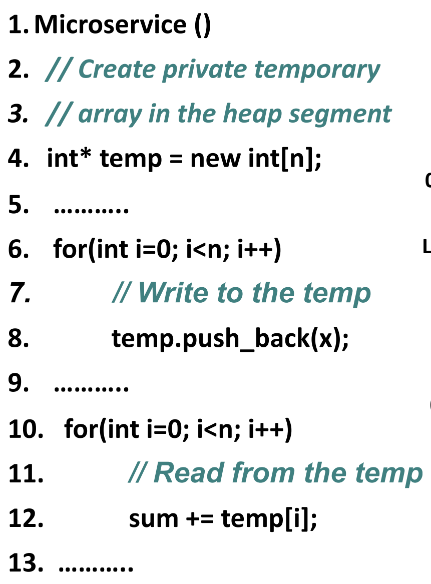


Figure 4.13. L1 MPKI of a single threaded CPU vs RPU with different batch sizes (32, 16, 8, 4).

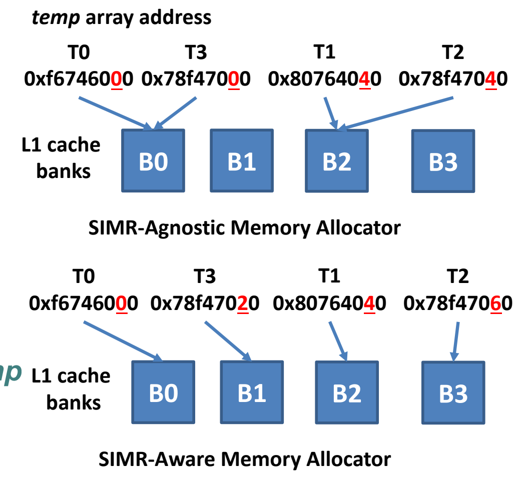
overall number of L1 accesses as well as the number of misses. As the batch size decreases, the coalescing efficiency is reduced.

On the other hand, some microservices, like *HDsearch-leaf* and *Textsearch-leaf*, have high L1 MPKI compared at a batch size of 32. These are data-intensive services, exhibiting a larger intra-thread locality footprint due to divergent heap segment accesses and read-after-write temporary data. However, they show low MPKI when we throttle the batch size to 8 (see Figure 4.13). We have similar observations for TLB and memory system contention when applying batch size tuning. Therefore, we run all our microservices at a batch size of 32, except for these data-intensive services, which are executed with a batch size of 8. Thanks to sub-batch interleaving, running at this smaller batch size does not affect our execution unit utilization. Regardless of batch size, the RPU hardware is designed with 8 SIMT lanes, as such, an 8-thread batch can fully utilize the pipeline, even though amortization suffers versus a 32-thread batch. It is worth noting that, after inspecting the *HDsearch* source code, we find that we can reduce the L1 cache footprint of the workload by eliminating some unnecessary data copies and employing function fusion (similar to kernel/layer fusion in GPU and DL); however, we decided not to alter the program in our experiments.



```
1. Microservice ()
2.  // Create private temporary
3.  // array in the heap segment
4.  int* temp = new int[n];
5.   ……….
6.   for(int i=0; i<n; i++)
7.          // Write to the temp
8.          temp.push_back(x);
9.   ……….
10.  for(int i=0; i<n; i++)
11.         // Read from the temp
12.         sum += temp[i];
13. ……….
```

(a) A divergent heap segement accsses of temp data



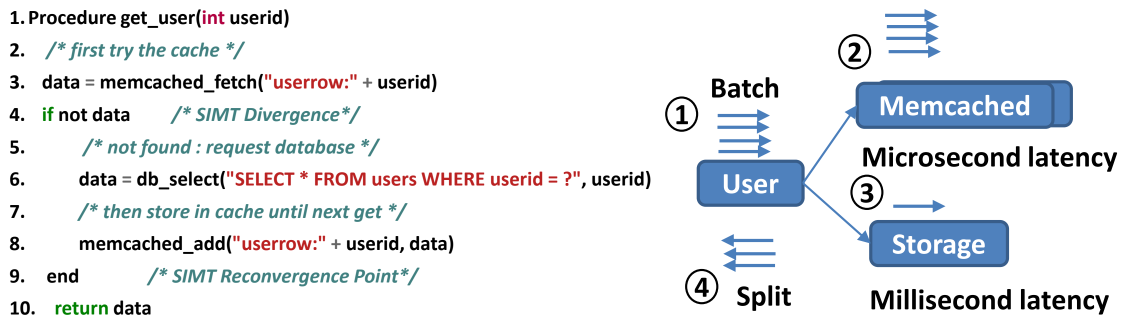
(b) SIMR-aware memory allocation assuming 4 banks 32B interleaving

Figure 4.14. SIMR-aware memory allocator.

Selecting the right batch size has many other factors, e.g. the request arrival rate and the system configuration. As widely practiced by data center providers [22], [30], an offline configuration can be applied to tune the batch size for a particular microservice. The time overhead to determine the correct batch size is well tolerated by data center providers and matches those used in Google and Facebook's batching mechanisms [22], [211].

SIMR-Aware Memory Allocation

Divergent accesses to the heap have the potential to create bank conflicts in the RPU's multi-bank L1 cache. Figure 4.14a depicts a frequent code pattern in our microservices. The program dynamically allocates a thread-private temporary array on the heap (line#4), fills the array with intermediate results in a linear fashion (line#8), and reads from this array to process thedata (line#12). The top section of Figure 4.14b shows how the default SIMR-agnostic CPU allocator may assign addresses to the *temp* array that result in significant bank conflicts. One solution for this is to change the address mapping of the heap segment [212] to interleave elements accessed by parallel threads, similar to our stack segment interleaving. However, this type of interleaving is ill-suited for heap accesses, which are less structured



```
Procedure get_user(int userid)
  /* first try the cache */
  data = memcached_fetch("userrow:" + userid)
  if not data      /* SIMT Divergence*/
      /* not found : request database */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid)
      /* then store in cache until next get */
      memcached_add("userrow:" + userid, data)
  end          /* SIMT Reconvergence Point*/
  return data
```

(a) Code snapshot for network divergence (b) Batch split

Figure 4.15. Batch split technique for control flow divergence when a path contains long network/storage blocking event.

than stack accesses. Another solution is to rely on hardware-based xoring hashing [65], [213], however our experiments show that it is ineffective to alleviate bank conflicts.

To this end, we address this problem by proposing a new SIMR-aware memory allocator that the RPU driver can provide as an alternative and overrides the memory allocator used by the run-time library through *LD_PRELOAD* Linux utility [214], [215]. Our proposed memory allocator, demonstrated in the bottom image of Figure 4.14b, avoids data interleaving for the heap segment. Instead, the key idea is to take into account that data are already interleaved every n bytes over L1 banks (n=32B in our baseline). Therefore, if we ensure that the start address of every new memory allocation per thread follows the condition *(start_address%(n*tid) = 0)*, then accesses to the private data structure will be conflict-free for all consecutive data accesses, as shown in Figure 4.14b. The overhead of this method is the unused few bytes at the start of each data allocation to ensure the alignment constraint (around 896 bytes for an 8-thread allocation). This memory fragmentation is amortized with large memory allocation sizes.

System-Level Batch Splitting

In the RPU, context switching is done at the batch granularity, either all threads in a batch are running or all the threads in the batch are switched out. When RPU threads are blocked due to an I/O event, the RPU driver groups the I/O interrupts and wakes the all the threads in the same batch at the same time to handle their interrupts and continue lockstep execution. However, requests with the same batch can follow different control paths, in which one path may be longer than the other. For memory and nanosecond-scale latencies, the paths synchronize the at the IPDOM reconvergence point. However, if one path contains significantly longer millisecond-scale latency (e.g., a request to storage or the network), this can hinder the threads on the other path, exaggerating the average latency. Figure 4.15a illustrates a frequently-used design pattern in microservice development, in which we cache the back-end storage accesses in a fast in-DRAM key-value store, like memcached (line#3 in Figure 4.15a). If the user request hits in the microsecond-scale latency memcached, the request returns immediately to the client (line#10); otherwise, it has to access the millisecond-scale storage, update the cache, and send the result back (lines#5-10). If the hit requests have to wait for the misses at the reconvergence point (line#9), then the storage latency will dominate the total average latency.

To avoid this issue, we propose, a batch splitting technique, as depicted in Figure 4.15b, in which we split the batch and allow multi-path execution [216] for hit and miss requests. That is, the batch is subdivided into two batches, one for the hit requests to continue execution beyond reconvergence point (④ in Figure 4.15b) and the other for blocked requests accessing the storage (③). It is worth noting that, in cycle-level multipath execution on GPUs [216]–[218], divergent paths still ultimately converge and resources are not freed until all paths are complete. In SIMR batch splitting, the fast completing path can be allowed to continue, and finish execution, while the slower blocked path is context switched out, freeing up resources for other requests.

A hardware-based timeout or software-based hint can be used to determine the splitting decision. Although batch splitting reduces control efficiency, as the miss requests will continue execution alone, we can still batch these orphan requests at the storage microservice and formulate a new batch to be executed with a full SIMT active mask. We believe there is a wide space of future work to analyze the microservice graph for splittng and batching opportunities.

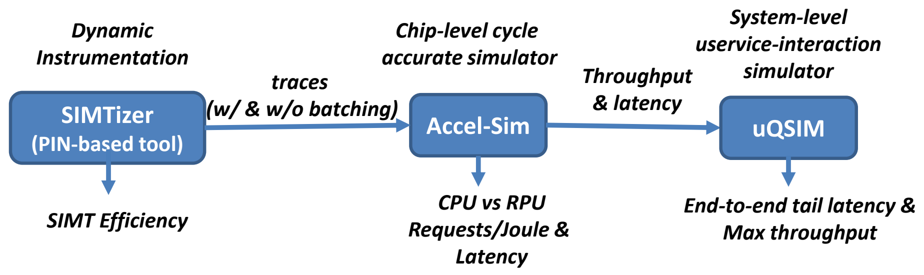

Dynamic Instrumentation
SIMTizer (PIN-based tool)
SIMT Efficiency
traces (w/ & w/o batching)
Chip-level cycle accurate simulator
Accel-Sim
CPU vs RPU Requests/Joule & Latency
Throughput & latency
System-level uservice-interaction simulator
uQSIM
End-to-end tail latency & Max throughput

Figure 4.16. End-to-End Experimental Setup

4.3 Experimental Setup

Workloads: We study a microservice-based social graph network as depicted in Figure 4.3, similar to the one represented in the DeathStarBench suite [27]. TextSearch, HDImageSearch, and McRouter are adopted from the usuite benchmarks [26], we use the input data associated with the suite. The microservices use diverse libraries, including c++ stdlib, Intel MKL, OpenSSL, FLANN, Pthread, zlib, protobuf, gRPC and MLPack. The post and user microservices are adopted from the DeathStarBench workloads [27] and social graph is from SAGA-Bench [219]. The microservices have been updated to interact with each other via Google's gRPC framework [220], and they are compiled with the -O3 option and SSE/AVX vectorization enabled. While the RPU can also execute other HPC/GPGPU applications that exhibit the SPMD pattern, like OpenMP and OpenCL, we only focus our study here on microservice workloads.

Simulation Setup: We analyze our RPU system over multiple stages and simulation tools. Figure 4.16 shows our end-to-end experimental setup. First, we analyze the SIMT efficiency of our microservice with an in-house x86 PIN [221]-based tool, named SIMTizer. The tool traces the dynamic control flow of CPU threads running in a batch with stack-based IPDOM reconvergence analysis [182], [207], and calculates the associated active mask and overall SIMT efficiency. SIMTizer traces the whole SW stack, including user code, libraries, and frameworks.

Second, we use the trace-driven, cycle-level Accel-Sim v1.1 [133] simulator and BookSim [222] for interconnect simulation to obtain chip-level throughput and service latency for the CPU vs. the RPU. We updated Accel-Sim's front-end to execute x86 traces generated by

Table 4.4. CPU vs RPU Simulated Configuration

Metric	**CPU**	**CPU SMT**	**RPU**
Core Pipeline	8-wide 256-entry OoO	8-wide 256-entry OoO	8-wide 256-entry OoO
ISA	x86-64	x86-64	x86-64
Freq	2.5 GHZ	2.5 GHZ	2.5 GHZ
#Cores	98	80	20
Threads/core	1	SMT-8	SIMT-32 (1 batch)
Total Threads	98	640	640
#Lanes	1	1	8
Max IPC/core	8	8	64 (issue x lanes)
ALU/Bra Exec Lat	1-cycle	1-cycle	4-cycle
#Stages (ALU-load)	9-12	9-12	14-18
L1 Inst/core	64KB	64KB	64KB
Reg File (PRF)/core	6KB	48KB	192KB
LSU (read/write)	128/64	128/64	128/64 (8x wide)
L1 Cache	64KB, 8-way, 3 cycles, 1-bank 32B/cycle	64KB, 8-way, 3 cycles, 8-bank 256BB/cycle	256KB, 8-way, 8 cycles, 8-bank 256B/cycle
L1 TLB	48-entry	64-entry	256-entry, 8-bank (32-entry/bank)
L2 Cache	512KB, 8-way, 12 cycles, 1-bank	512KB, 8-way, 12-cycles, 2-banks	2MB, 8-way, 20 cycles, 2-banks
L3 Cache	32MB, 16-way	32MB, 16-way	32MB, 16-way
DRAM	8x DDR5-3200, 200 GB/sec	10x DDR5-7200, 576 GB/sec	10x DDR5-7200, 576 GB/sec
Interconnect	9x9 Mesh	11x11 Mesh	20x20 Crossbar
OoO entries/thread	256, 8-wide	32, 1-wide	256, 8-wide
L1 capacity/thread	64KB	8KB	8KB
TLB entries/thread	48	8	8
L1B/cycle/thread	32B/cycle	32B/cycle	32B/cycle
memBW/thread	2 GB/sec	0.9 GB/sec	0.9 GB/sec

SIMTizer. CISC instructions with memory operands are broken down to multiple RISC-like instructions with separate loads and stores [223]. Further, Accel-Sim's performance model has been extended to model a CPU-like pipeline with superscalar, OoO issue. Table 4.4 lists the simulator configuration for CPU vs. RPU. We model many-core x86-based single-threaded CPU similar to the ones found on the market today and used in data centres [149]–[152]. We also model an 8-way simultaneous multi-threading CPU (SMT8), to reflect the highest SMT degree found in the market today from IBM POWER9 [154].

We ensure both CPU and RPU have the same pipeline configuration, and frequency. For SMT8, we maintain the same number of total threads and memory resources/thread vs RPU (see the last four entries in Table 4.4). Cache latency is calculated based on CACTI v7.0 [224]. The multi-bank caches and MCU increase the L1/L2 hit latency from 3/12 cycles in the CPU to 8/20 cycles in the RPU. For other execution units, the ALU/Branch execution latency is increased to 4 cycles in the RPU to take into account the extra wiring and capacitance of adding more lanes [161] and the majority voting circuit. We assume an

Table 4.5. Per-component area and peak power estimates

Component	Area				Peak Power			
	CPU		RPU		CPU		RPU	
	mm^2	% Core	mm^2	% Core	Watt	% Core	Watt	% Core
Fetch&Decode	0.27	24.3	0.3	4.3	0.39	15.6	0.4	3.6
Branch Prediction	0.01	0.9	0.01	0.1	0.02	0.8	0.02	0.2
OoO	0.11	9.9	0.17	2.4	0.85	34	1.45	12.9
Register File	0.14	12.6	2.52	35.8	0.49	19.6	4.26	38
Execution Units	0.25	22.5	2.31	32.8	0.34	13.6	2.51	22.4
Load/Store Unit	0.07	6.3	0.34	4.8	0.13	5.2	0.41	3.7
L1 Cache	0.04	3.6	0.22	3.1	0.09	3.6	0.2	1.8
TLB	0.02	1.8	0.08	1.1	0.06	2.4	0.4	3.6
L2 Cache	0.2	18	0.71	10.1	0.13	5.2	0.24	2.1
Majority Voting	0	0	0.03	0.4	0	0	0.05	0.4
SIMT Stack	0	0	0.02	0.3	0	0	0.01	0.1
MCU	0	0	0.02	0.3	0	0	0.03	0.3
L1-Xbar	0	0	0.31	4.4	0	0	1.23	11
Total-1core	1.11		7.04		2.5		11.21	
	mm^2	% Chip	mm^2	% Chip	Watt	% Chip	Watt	% Chip
Total-Allcores	108.8	77.2	140.8	81	245	72.5	224.2	73.7
L3 Cache	7.82	5.5	7.82	4.5	0.75	0.2	0.75	0.2
NoC	9.78	6.9	1.72	1	36.52	10.8	7.02	2.3
Memory Ctrl	14.64	10.4	23.59	13.6	6.85	2	19.27	6.3
Static Power					49	14.5	53	17.4
Total Chip	141		173.9		338.1		304.2	

idealistic cache coherence protocol for the CPU, with zero traffic overhead, in which atomics are executed as normal memory loads in private cache, whereas, in RPU, atomic instructions bypass private caches and execute at shared L3 cache.

Third, to study batching effects on a large scale and system implications with context switching, queuing delay, and network/storage blocking, we harness uqsim [225], an accurate and scalable simulation for interactive microservices. The simulator is configured with our social graph network along with the latency and throughput obtained from Accel-Sim simulations to calculate system-wide end-to-end tail latency.

Energy&Area Model: We use McPAT [226], and some elements from GPUWattch [187] to configure the CPU and RPU described in Table 4.4, to estimate per-component area, peak power and dynamic energy. For the RPU, we consider the additional components and augmentation required to support SIMT execution described in Figure 4.6. The majority voting circuitry is modeled as a CAM structure (32-way comparator) to count the taken and non-taken results and a reduction tree to calculate the most selected destination address. The SIMT stack is modeled as 1KB memory structure [183]. A 2x 32-way CAM structure is used to model the memory coalescing units [187], and the RAT, ROBs, and uop buffers are extended to include the 4-byte active mask and its associated logic.

Table 4.5 shows the calculated area and peak power for the RPU and single-threaded CPU at 7-nm technology [227]. The CPU's frontend+OoO area and power overhead are roughly 40% and 50% respectively, which are aligned with modern CPU designs [12]. The table shows that the RPU core is 6.3x larger and consumes 4.5x more peak power than the CPU core; however, the RPU core supports 32x more threads. In the RPU core, most of the area is consumed in the register file and execution units, 68% of the area vs. 35% in the CPU. The additional overhead of the RPU-only structures consume 11.8% of the RPU core. Most of this overhead comes from the 8x8 crossbar that connects the L1 banks to the LD/ST queues. To support SMT-8 in the CPU, 14% area and power increase per core is required (not shown in the table for brevity). In Section 4.4, we use the per-access energy numbers generated from our McPAT analysis with the simulation results generated by Accel-Sim to compute the runtime energy efficiency of each workload (Figure 4.17).

4.4 Experimental Results

4.4.1 Chip-Level Results

Figure 4.17 and Figure 4.18 show energy efficiency (Requests/Joule) and service latency of RPU and CPU-SMT8 normalized to single threaded CPU. All the hardware executes the same number of requests (2400). On average, the RPU can achieve 5.6x higher energy efficiency compared to CPU, while still coming within 1.35x of its service latency, with the worst service latency of 1.7x at *HDSearch-midtier*. Overall, the RPU's service latency remains under the 2x higher latency limit defined by data center providers [170]. The main causes of RPU's energy-efficiency are: (1) reducing the number of issued instructions by a factor of 30x, amortizing the frontend and OoO dynamic energy overhead that accounted for up to 70% in some scalar heavily-integer microservices, (2) generating 4x less traffic on average, therefore decreasing the memory energy consumption, and (3) running 6x more requests at almost the same service latency vs. the CPU, and thus amortizing the static energy. The *HDSearch* and

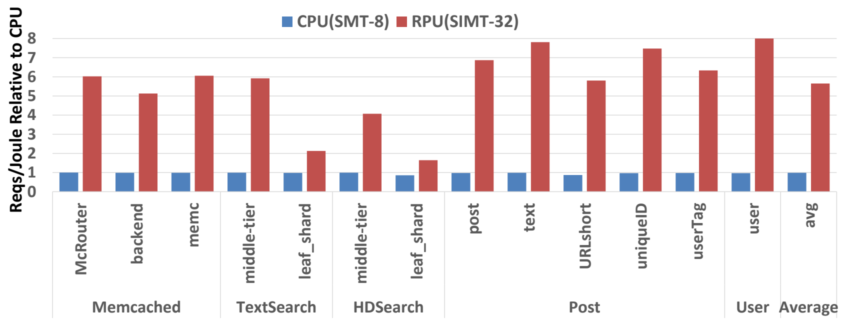


Figure 4.17. RPU and CPU-SMT8 energy efficiency (Requests/Joule) relative to single threaded CPU (higher is better)

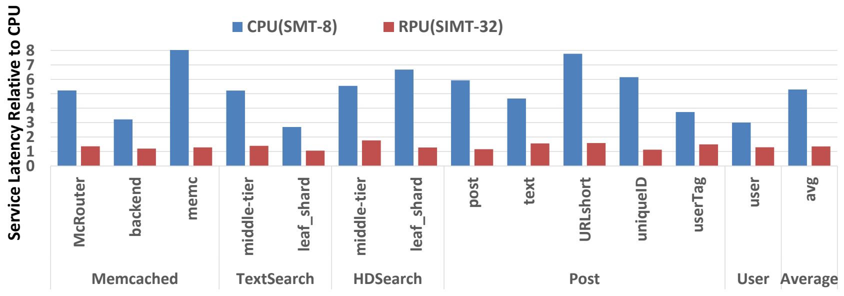


Figure 4.18. RPU and CPU-SMT8 service latency relative to single threaded CPU (lower is better)

TextSearch microservices exhibit less energy-efficiency than the average. These workloads run at a smaller batch size, and the frontend+OoO only accounts for 33% of the CPU's energy.

On the other hand, CPU-SMT8 only improves energy efficiency by 5% at a 5x higher service latency cost. This is because the number of issued instructions and the generated accesses are the same as in single threaded CPU. Further, SMT8 partitions the front-end resources per thread and causes cache serialization of stack segment accesses and shared heap variables, hindering service latency, whereas RPU avoids all these issues through SIMT execution.

The main causes of our 1.35x higher service latency in the RPU are three-fold. First, the control SIMT efficiency of some microservices like *text* and *Textsearch* is below 90% (see Table 4.10) in which the RPU serializes the divergent paths and increases service latency. Second, when CPU threads run consecutively, they prefetch some shared data to the L1 cache for the incoming threads running on the same core. In the RPU, many threads are run in parallel and incur these compulsory misses at the same time. Third, the L1 access latency of the RPU is longer (3 vs 8 cycles) as a result of a larger L1 cache size, the MCU and multi-bank arbitration.

Sensitivity Analysis

- **Sub-batch interleaving**: In the CPU, IPC per thread is limited, with an average IPC of 1, similar to those reported in data center studies [30], [143]–[145]. In the RPU, and thanks to sub-batch interleaving, we are able to improve our IPC utilization up to 4x by issuing threads over multiple cycles to the SIMT lanes. Although we reduced the number of SIMT lanes by 4x with sub-batch interleaving (i.e., from 32 to 8 lanes), we only noticed 2% performance loss on average compared to full width SIMT lanes

- **Moving atomics to L3**: We did not notice slow down from moving atomics to L3 cache in the RPU as our microservices exhibit low atomic/locks per instructions.

- **SIMR-aware heap allocation**: our SIMR-aware heap segment improves the L1 cache throughput for frequent divergent heap segments in *HDSearch* and *TextSerach*, where a 1.9x higher throughput was achieved versus the RPU SIMR-agnostic heap allocations.

- **Majority voting**: Majority voting optimizes the branch prediction for the common control flow (92% of the time threads traverse the same control flow). Still, the 8% control divergence causes some threads to have different predictions than they would with a per-thread prediction (i.e., as in CPUs). Since we predict next PC per entire batch, we will always have misprediction for the divergent threads of the other path. Majority voting mitigates the flushes caused by these inevitable branch mispredictions by optimizing for the common control flow, and thus improving overall energy efficiency. The Majority voting has little impact on performance, instead it aims to optimize power consumption. Improving the SIMT branch prediction such that it can predict the next PC along with the associated active mask is an interesting area of future research.

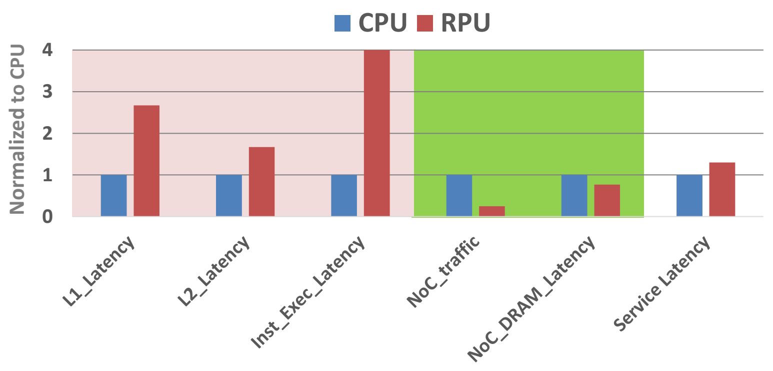


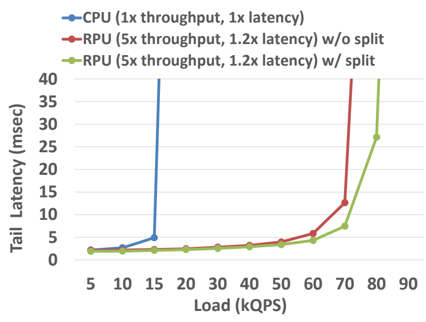
Figure 4.19. Metrics that contribute to total service latency.

Service Latency Analysis

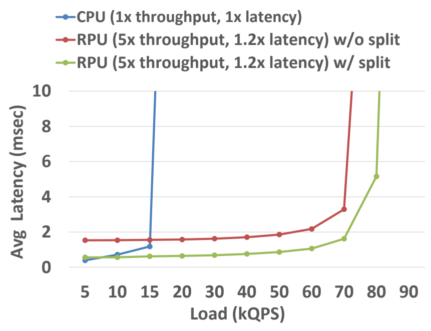
Despite our higher L1 access (2.3x), ALU and branch execution latency (4x), and control divergence (8%), some microservices are still able to achieve service latency close to the CPU, and on average only 1.35x higher latency. This is because memory coalescing has reduced the on-chip memory traffic, alleviating contention and minimizing the memory latency. Figure 4.19 depicts several metrics that explain the relatively little increase in service latency for the RPU. The average network on chip (NoC) latency has been reduced by 1.33x because 4x less traffic is generated. The RPU's memory coalescing and single-hop crossbar interconnect both combine to offset the latency increases in instructions and cache hits.

GPU Performance

We also run our experiments on an Ampere-like GPU model [228] with the same software optimization as the RPU (e.g., stack memory coalescing and batching) and assuming that the GPU supports the same CPU's ISA and system calls. For the sake of brevity, we did not plot the per-app results in the figures. On average, the GPU achieves 28x higher energy efficiency than the CPU but at 79x higher latency. This high latency is unacceptable for QoS-sensitive data center applications [170], [172], [229]. These results are expected and aligned with previous work executing server workloads on GPUs [156].



(a) End-to-end 99% tail latency



(b) End-to-end average latency

Figure 4.20. End-to-end tail and average latency for CPU-based system vs RPU-based system with and without batch split.

4.4.2 System-Level Results

Figure 4.20 shows the system-level, end-to-end 99% tail and average latency for CPU-based system and RPU-based system with and without our batch splitting technique described in Section 4.2.2. We scale the QPS load until reaching the highest maximum throughput at acceptable QoS and the system saturates. We configure uqsim with the end-to-end *User* microservice scenario passing from *Web Server* to *User* to *McRouter* to *Memcached* and *Storage* in Figure 4.3.

We simulate three CPU server machines with 40 cores, where each microservice runs on its own server node. We assume a 90% hit rate of *Memcached* with 100, 20, 25, 1000 and 60 microseconds latency for *User*, *McRouter*, *Memcached*, *Storage* and network respectively. In the RPU configuration, we replace the CPU servers with RPU machines consuming the same power budget, i.e. assuming 5.2x higher Requests/Joule and 1.2x higher latency as were obtained from chip-level experiments for these services. Request batching is employed for *memcached* in the CPU configuration for *epoll* system call to reduce network processing, as is the common practice in data centers [225]. To focus our study on processing throughput, we assumed unlimited storage bandwidth for both CPU and RPU configurations.

From analyzing the end-to-end results in Figure 4.20, we can make the following observations. First, the RPU (with batch split) can achieve 5x higher request throughput per Joule compared to the CPU with almost the same tail and average latency. Second, the batching formulation time is amortized and incurs negligible overhead at both low and high traffic

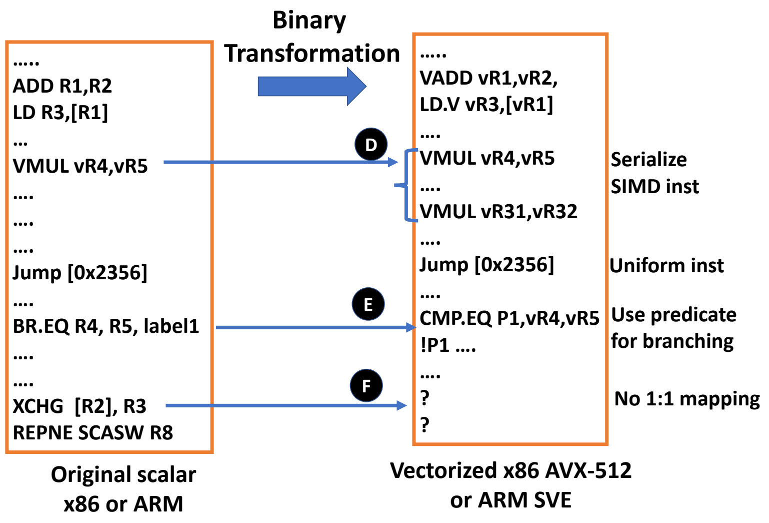


Figure 4.21. Potential binary transformation of a scalar binary to a vector version

load. This is due to the fact that CPU system employs batching already for memcached. Third, without batch splitting on millisecond-scale storage accesses, the RPU exhibits higher average latency than the CPU, as blocked threads are waiting on a reconvergence point for the others that access the storage. However, RPU without batch splitting is still able to attain acceptable tail latency. Although tail latency is more important than average latency for QoS measurements, the batch splitting technique can be beneficial to ensure predictable responsive time when unpredictable high latency episodes occur in large online services [229].

4.5 Discussion

4.5.1 RPU vs CPU's SIMD

A possible alternative to the RPU would be recompile scalar CPU binaries for execution on the CPU's existing SIMD units, e.g., x86 AVX [230], [231] or ARM SVE [232]. Each request could be mapped to a SIMD lane, amortizing the front-end overhead, leveraging the latency optimizations of the CPU pipeline, and executing uniform instruction on the scalar units [233]. Such a transformation could be done using a SPMD-on-SIMD compiler, like Intel ISPC [233], or at the binary-level, as depicted in Figure 4.21. However, this solution has three primary shortcomings. First, it requires a complete recompilation of the microservice code, libraries, and OS system calls. Second, SIMD units on contemporary CPUs are designed to accelerate computationally-dense inner loops. The memory system and vector ISA are not optimized for the branch- and memory-heavy microservices we focus on in the RPU. As a result, energy-efficiency and service latency will be negatively affected. For instance, we need to serialize existing SIMD instructions in the scalar binary (D in Figure 4.21), predicate computation that cannot take advantage of branch prediction (E), and the fact that there are 2-3x more scalar units than SIMD units [149], [150] on existing CPUs, which will go unused if the code could be fully vectorized. Finally, many existing scalar instructions lack a 1:1 mapping with any vector instruction (F), e.g., complex string manipulation, atomic and OS operations. Based on a manual investigation in x86 ISA [230], there are 129 AVX instructions, and 463 scalar instructions, thus only a maximum of 27% of the scalar instructions are represented in the vector ISA.

4.5.2 Multi-threaded vs Multi-process Services

Our proposed SIMR system focuses on multi-threaded services, which are widely used in data centers [144], [234]. However, the rise of serverless computing has made multi-process microservices more common [27], [235]. In multi-process services, the separate virtual address spaces can cause both control flow and memory divergence, even if the processes use the same executable and read the same data, which also causes cache-contention issues on contemporary CPUs. We believe that with user-orchestrated inter-process data sharing and some modifications to the RPU's virtual memory; these effects can be mitigated. However, since the contemporary services we study are all multi-threaded, we leave such a study as future work.

4.5.3 Security Implications

The grouping of concurrent requests for SIMT execution may enable new vulnerabilities. For instance, a malicious user may generate a very long query that could affect the QoS of other short requests or leak control information. Such attacks can be mitigated in our input size-aware batching software by detecting and isolating maliciously long requests, as described in Section 4.2.2. Another security vulnerability is the potential for parallel threads to access each other's stack data (exploiting the fact that threads' stack data are adjacent in the physical space). However, as described in Section 4.2.2, the RPU's address generation unit is able to identify inter-thread stack accesses and throw an exception if such sharing is not permitted.

4.5.4 GPGPU Workloads on RPU

RPU can also execute other HPC, GPGPU, and DL applications that exhibit the SPMD pattern, written in OpenMP, OpenCL, or CUDA. GPUs have been shown to be 2-5x more energy efficient than CPUs [236]–[239], thanks to its simpler In-Order pipeline and software-managed caches. However, this comes at the cost of easy-to-program. Developers need to rewrite the code in GPGPU programming language and make a heroic effort to get the most out of GPU's compute efficiency [128], [240]. Recently, and to achieve high efficiency in the lack of HW-support OoO scheduling, Nvidia has written its back-end libraries in hand-tuned machine assembly to improve instruction scheduling [133] and proposed complex asynchronous programming APIs [241] to hide memory latency via prefetching. In CPUs, the HW-support OoO with large instruction window relieve this burden from the programmers.

On the other hand, we believe that RPU takes the best of both worlds. It can execute GPGPU workloads with the same easy-to-program CPU interface and still get as close as to the GPU's energy efficiency. By amortizing the frontend and OoO overhead, the RPU can reduce the energy efficiency gap between CPU and GPU, becoming a first-class engine for HPC/GPGPU as well as microservices workloads. We leave studying such an argument and evaluate the achievable energy efficiency of GPGPU workloads on RPU for future work.

4.5.5 RPU vs GPU Terminology

RPU and GPU are SIMT-based hardware. However, through this thesis, we have used different hardware terminology. Table 4.6 compares between Nvidia's GPU and our RPU terminology.

Table 4.6. GPU vs RPU Terminology

GPU	**RPU**
Grid/Thread Block	SW Batch
Warp	HW Batch
Thread	Thread/Request
Kernel	Service
GPU Core / Streaming MultiProcessor (SM)	RPU Core / Streaming MultiRequest (SM)
Warp Scheduler	Batch Scheduler
SIMT	SIMR
CUDA core	Execution Lane

4.6 Summary

Data center computing is experiencing an energy-efficiency crisis.. Aggressive OoO cores are necessary to meet tight deadlines but waste excessive energy and limit the number of threads that can be packed into one core. However, modern productive software has inadvertently produced a solution hardware can exploit: the microservice. By subdividing monolithic services into small pieces and executing many instances of the same microservice concurrently on the same node, parallel threads execute similar instruction controlflow and access similar data. We exploit this fact to propose our Single Instruction Multiple Request (SIMR) processing system, comprised of a novel Request Processing Unit (RPU) and an accompanying SIMR-aware software system.

The RPU adds Single Instruction Multiple Thread (SIMT) hardware to a contemporary OoO CPU core, maintaining single threaded latency close to that of the CPU. As long as SIMT efficiency remains high, all the OoO structures are accessed only once for a group of threads, and aggregation in the memory system reduces accesses. Complimenting the RPU, our SIMR-aware software system handles the unique challenges microservice + SIMT computing by intelligently forming/splitting batches and managing memory allocation. Across 13 microservices, our SIMR processing system achieves 5.6x higher Requests/Joule, while only increasing single thread latency by 1.35x. We believe the combination of OoO and SIMT execution opens a series of new directions in the data center design space, and presents a viable option to scale on-chip thread count in the twilight of Moore's Law

5. RELATED WORK

This chapter summarizes and contrasts the work done in this dissertation against related work in GPU simulations, NUMA-aware management for CPUs and GPUs, and online web services/microservices acceleration. Section 5.1 discusses work relating to Accel-Sim. Section 5.2 details work related to LADM. Section 5.3 gives overview on related work to SIMR in literature.

5.1 GPU Simulation

As Table 2.1 illustrates, there has been extensive work done in GPU simulation over the last decade. Xun et al. [53] model GPU Kepler architecture [242] that is integrated with the Multi2sim simulation infrastructure [54]. The new simulator was validated using applications from the CUDA SDK. Power et al. [243] propose the Gem5-GPU simulator which models detailed CPU-GPU interaction. Beckmann et al. [48], [49] introduce a Gem5 GPU model, that simulates AMD's GCN architecture. Barrio et al. [244] propose ATTILA, which is a trace-driven GPU simulator that models the graphics pipeline for the GT200 architecture. Gubran et al. [245] update GPGPU-sim to execute graphics workloads. Sniper [246] is a multithreaded CPU simulator that can execute in trace- or execution-driven mode using PIN-based callbacks to simulate instructions without emulation. In this vast landscape, Accel-Sim is the only open-source framework, explicitly designed for validation, that can simulate NVIDIA SASS instructions and accurately model contemporary NVIDIA GPUs.

Stephenson et al. [247] introduce SASSI, a low-level assembly-language instrumentation tool for GPUs. Many previous works [248]–[253] aim to estimate the execution time of GPGPU applications throughout source-level analysis, however they do not perform detailed performance modeling for architecture research. Nowatzki et al. [254] show that GPGPU-Sim is inaccurate in modeling some in-core parameters and connection delays. Hongwen et al. [255] argue that GPGPU-Sim has a very weak L1 cache model and they suggest enhancements to improve the L1 cache throughput. Jain et al. [47] demonstrate a large gap in GPGPU-Sim 3.x's memory system when correlating against Pascal hardware. Prior work [256]–[258] propose a methodology for validating CPU simulation models against real hardware. Numerous works have done micro-benchmarking to characterize aspects of GPU architecture for Tesla GT200 [42], Fermi [43], Maxwell [44], and recently Volta [45]. They demystified the L1/L2 cache's associativity and replacement policy. However, none of these works have investigated the L1 streaming cache, sectoring of the L1/L2 caches and the interaction of the memory coalescer with caches or the cache write policy.

5.2 NUMA-aware Management for CPUs and GPUs

A number of researchers [98]–[100] have explored disintegrating multi-core CPUs into smaller chips in order to improve manufacturing yield. In a multi-GPU context, past work [10], [102], [120] investigated similar multi-socket and MCM NUMA GPU designs to scale GPU performance beyond a single socket. We have discussed their approaches in details throughout this thesis and compare their results with LADM. Baruah et al. [123] propose hardware-software support for page migration in multi-GPU shared-memory systems. Milic et al. [10] propose dynamic, phase-aware interconnect bandwidth partitioning. They also dynamically adapt L2 caching policy to minimize NUMA effects. These works employ reactive runtime solutions whereas we apply a low-overhead proactive approach.

Young et al. [120] propose a DRAM-cache with optimized hardware coherence for multi-GPU systems. Xiaowei et al. [134] propose a customized L2 cache coherence protocol for hierarchical multi-chiplet multi-GPU systems. These cache coherence protocols are orthogonal to our work and can be applied on top of LADM for further performance improvement.

While significant work has been done to optimize weak-scaling performance using MPI + GPUs (where each rank controls a GPU operating on a relatively isolated partition of data [259], [260]) or via the OpenCL runtime driver [261], [262]. However, transparently achieving *strong scaling* on NUMA-GPU systems with diverse sharing patterns is still an open problem, which we aim to address in this work.

Prior work on locality-aware threadblock scheduling in single GPU contexts has either not used static analysis [131], [263], [264] or performed a subset of the analysis done by LADM [125], [265] simply because the placement of data has not been an objective. Handling page alignment, the effect of remote caching, and matching competing access patterns to data structures are all issues that arise in the NUMA context that are not addressed in prior work on threadblock scheduling for cache locality. It is difficult to provide a fair quantitative comparison to these works, as it requires us to fill-in-the-blanks on how the techniques would be applied to NUMA-GPUs.

Several works [121], [122], [266], [267] have provided batching and reactive prefetching to improve UVM performance in single GPU systems. LASP can be extended to efficiently support oversubscribed memory by proactively placing the next page where it is predicted to be accessed, avoiding page-faulting overheads. Using the locality table information, the pages that are already accessed by finished threadblocks and will not be used again, can be evicted and replaced with the new pages *proactively*.

Compiler-assisted index analysis has been used in CPUs and GPUs to perform affine loops transformation in order to: (1) improve locality via data tiling within a single-GPU machine [268]–[270], and (2) automatically parallelize serial code on parallel machines [103], [271]–[273]. However, these works perform source-to-source transformation and do not provide any runtime decisions on *how* to efficiently schedule the threads. Furthermore, prior work on GPU static analysis does not exploit all the locality patterns identified by LADM. In this work, we extend single thread index analysis to be threadblock-centric for the NUMA-GPU domain.

It is worth mentioning that, with modifications to account for threadblock motion and inter-thread sharing, a polyhedral framework [270], [274], [275] could be used in place of LADM's index analysis. However, we believe that LADM‘s simpler and effective index-based analysis increases the likelihood it will be adopted in contemporary GPU compilers (e.g. NVCC [276]). Either way, the choice of compiler infrastructure used is orthogonal to the datablock analysis proposed in this thesis.

Data placement has been a focus of CPU research in OpenMP NUMA systems. Solutions include adding new OpenMP language primitives which are explicitly used by the programmer [277]–[280], compiler-assited page migration [281], [282] or reactively changing the virtual page size [283]. Although thread scheduling is a concern in CPU-NUMA systems, the focus is largely on workload balancing via advanced work stealing algorithms [284] or avoiding cache thrashing [285], but not to ensure memory page locality. In this work, we coordinate both data placement and thread scheduling to exploit various locality patterns of massively multithreaded multi-GPU systems.

5.3 Webservices Acceleration on GPUs

Server Workloads on GPUs: The most closely related work to SIMR are [155], [156], [163]. Agrawal et al. [156] proposed to run data center server workloads, SPEC-Web benchmarks, in lock-step execution on GPUs to expliot request similarity. While achieving significant energy efficiency, the authors had to rewrite the workloads from PHP to CUDA. Similarly, Hetherington et al. [155], [286] run memcahced workload on a GPU. The longer request latency, system calls support, and limited programmability are hinder the adoption of GPUs for general data center workloads with limited intra-request parallelism. Agrawal et al. [163] study the SIMT efficiency of SEPC-web workloads and show that it contains promising control and memory efficiency that can be executed on SIMT hardware.

System Calls on GPUs: Previous studies have explored supporting system calls on GPUs, including, file system [157], networking [158], and generic system calls [159]. In all these studies, the actual system calls are still executed on the CPU, as GPU cannot run OS the layer and does not have direct access to system I/Os. Thus, they had to build an event-handling wrapper framework to execute system calls on CPUs, which increased the request latency due to frequent PCIe communication between CPU and GPU. NVIDIA has proposed GPUDirect technology [160] which aims to optimize data movement of GPUs to I/Os, by copying the data directly from the network card and storage to GPU memory, however, the actual system call processing is still being executed on the CPU side.

SIMT+OoO Execution: Kalathingal et al. [287] proposed dynamic inter-thread vectorization architecture to leverage the implicit similarity that exists across SMT threads when running SPMD Rodinia applications. Tino et al. [179] introduced an out of order pipeline for SIMT hardware to optimize OpenMP workloads. Similar threads are detected and grouped dynamically with hardware support. Previous work [288]–[292] explored adding light-weight out-of-order execution in GPUs to further improve memory latency hiding. Our work is fundamentally different in that we start with an aggressive OoO CPU design, then adding GPU-like SIMT elements as necessary to improve energy-efficiency. This approach frees the RPU from the constraints of the GPU programming model, introducing several new challenges we must address to efficiently execute general-purpose pre-compiled microservices. Examples include handling network divergence (Section 4.2.2), optimizing microservice batches (Section 4.2.2), supporting more exotic features such as inter-thread stack sharing (Section 4.2.2). In addition, prior GPU+OoO approaches still relied on massive fine-grained multithreading and focus on throughput, whereas the RPU has significantly fewer threads and balances throughput and latency, addressing the unique challenges in the memory system. Examples include low-latency memory coalescing (Section 4.2.1) and SIMR-aware memory allocation (Section 4.2.2). Furthermore; none of these prior work has made the connection between SIMT and microservices.

Microservices Acceleration: Previous work [28], [29], [293], [294] have explored using hardware to accelerate microservices, with a focus on remote procedure calls [28], [29], [294], and network data transformation [293]. These proposals are orthogonal to our work and could be applied on top of the RPU. These works focus on helping the CPU remove isolated bottlenecks, whereas the RPU focuses on a full system solution that is meant to replace the CPU.

6. CONCLUSIONS AND FUTURE WORK

This chapter concludes the dissertation and proposes potential future work based on its findings.

6.1 Conclusions

SIMT-based accelerators, like GPUs and RPUs, are promising solutions to achieve significant *energy efficiency* in the data centers while still preserving *programmability.* In this dissertation, I tried to address the following three challenges: (1) how do we overcome the non-uniform memory access overhead for next-generation multi-chiplet GPUs in the era of DL-driven workloads?; (2) how can we improve the energy efficiency of data center's CPUs in the light of microservices evolution and request similarity?; (3) How to build and accurate and extensible SIMT simulation that can keep up with industrial changes?

First, modeling an accurate simulator for contemporary accelerators plays a crucial role in the computer architecture field. An old inaccurate architecture model may lead to unrealistic issues or incorrect conclusions that are irrelevant to industrial designs. Since GPU is becoming the defacto standard for ML training, I rigorously correlated the commonly-used GPGPU-Sim simulator with contemporary hardware. Conclusions from this analysis show that the absolute error between hardware and simulation performance is relatively high (between 22% and 105%). I concluded that the two main sources of simulator inefficiency are: *(i)* inaccurate memory system modeling, and *(ii)* simulating old machine ISAs. To improve the quality of GPU research produced by the academic research community, I have developed *Accel-Sim* [133], [295], a new GPU simulation framework to help solving the problem of keeping simulators up-to-date with contemporary designs and enabling GPU-based deep learning research with tensor cores and multi-GPU systems.

Second, thanks to high levels of inherent parallelism, many GPU workloads will be able to strongly scale performance, if large enough GPUs can be built. However, due to the physical limitations of chip and interconnect technologies, GPUs built with enough resources to leverage this abundant parallelism will have to overcome significant NUMA effects. This thesis describes a locality aware data management system (LADM) [298] designed to transparently overcome the NUMA effects of future GPUs by combining static analysis with hardware data placement, thread scheduling, and cache insertion policies LADM can decrease inter-GPU memory traffic by 4× and improve system performance by 1.8× across a range of workloads with varying locality. LADM demonstrates that intelligent coordination of thread scheduling and data placement can offset the need for expensive GPU interconnect technologies in the future.

Third, data center computing is experiencing an energy-efficiency crisis. Aggressive OoO cores are necessary to meet tight deadlines but waste excessive energy and limit the number of threads that can be packed into one core. However, modern productive software has inadvertently produced a solution hardware can exploit: the microservice. By subdividing monolithic services into small pieces and executing many instances of the same microservice concurrently on the same node, parallel threads execute similar instruction controlflow and access similar data. We exploit this fact to propose our Single Instruction Multiple Request (SIMR) processing system, comprised of a novel Request Processing Unit (RPU) and an accompanying SIMR-aware software system.

The RPU adds Single Instruction Multiple Thread (SIMT) hardware to a contemporary OoO CPU core, maintaining single threaded latency close to that of the CPU. As long as SIMT efficiency remains high, all the OoO structures are accessed only once for a group of threads, and aggregation in the memory system reduces accesses. Complimenting the RPU, our SIMR-aware software system handles the unique challenges microservice + SIMT computing by intelligently forming/splitting batches and managing memory allocation. Across 13 microservices, our SIMR processing system achieves 5.6x higher Requests/Joule, while only increasing single thread latency by 1.35x. We believe the combination of OoO and SIMT execution opens a series of new directions in the data center design space, and presents a viable option to scale on-chip thread count in the twilight of Moore's Law.

6.1.1 Other Collaboration Work

To further improve GPU simulation methodology, I proposed, in collaboration with researchers from various universities, *Principle Kernel Analysis* [296] and *Accel-Wattch* [297] which address simulation time and power modeling respectively. In [296], we demonstrate how simulating century-long scalable ML workloads, like MLPerf benchmarks, can be reduced to a matter of hours with error rates that are in line with the baseline simulator.

6.1.2 Potential Impact

The GPU simulation framework, *Accel-Sim*, opens up a rich new design space in system-level GPU-based research and reduce the accuracy gap between industrial and academic simulators on an ongoing basis, increasing the potential impact of academic research. In addition, the simulator can be configured to simulate emerging SIMT-based accelerator, like RPU, with x86 traces for data center microservices workloads. Based on our survey, Accel-Sim is considered the most widely used and highly validated GPU simulation framework in the academia. As it has been shown in a recent work from NVIDIA Research on building their in-house GPU simulator [299], Accel-Sim demonstrates accuracy comparable to simulators used in industry. Therefore, and thanks for its flexibility and usability, Accel-Sim is widely used by national labs and emerging industrial start-ups.

Further, my proposed LADM solution can transparently be used to run CUDA programs as-is on current multi-GPU multi-chiplet systems, achieving strong GPU scaling without burdening the programmer and enabling continuous performance scaling in the twilight of Moore's law. Finally, and in the light of microservice era, the new RPU-based system is a compelling approach to design more cost-effective data centers by exploiting request similarity and reducing the growing CPU's front-end power consumption. This dissertation will help in building our tomorrow's hardware to be more programmable, scalable, energy-efficient, reducing carbon emissions.

6.2 Future Research Directions:

There are various research directions that can be built on the top of this dissertation: (1) identifying multi-trillion model training bottlenecks, (2) RPU prototyping, and (3) addressing the killer microseconds devices.

6.2.1 Enabling Efficient Multi-Trillion Model Training

Identifying Bottlenecks with Heterogeneous Memory

A key observation from modern GPU generations is that memory capacity do not scale at the same pace as compute resources. As a result, many GPU workloads are limited by memory capacity. In [300], I demonstrated how the memory capacity can affect the ML hardware functionality and limit the model and batch size that can run on the system. As of today, the largest natural language understating model is Google's Gshard model with 600B parameters. In order to achieve human's level accuracy, the ML experts aim to build a multi-trillion parameter model which requires a significant amount of memory capacity. Employing heterogeneous memory seems to be a compelling solution to improve memory capacity scalability and has been adopted by several ML hardware vendors and startups. By integrating multiple levels of heterogeneous memory modules, training giant multi-trillion models in a cost-effective way is possible. This is achieved by saving the model weights in a large-capacity low-speed memory (e.g., DRAM or SSD storage), and proactively fetch the next active layer's weights into a small-capacity high-speed memory (e.g., HBM or SRAM), i.e., overlapping compute with memory access. In this research, I set out to build an analytical model and a new tool that quantitatively evaluate the hardware's ability to hide the latency and identify performance bottlenecks, taking into account the various system configuration, including batch size, back-end memory bandwidth, I/Os, network topology, layer size, model type (GPT-like dense layer or Google's sparse MoE layer), and overall system compute efficiency. Such tool may be very beneficial for the ML scientists in a way that they can tune the above parameters to find the best configuration, hence reaching the highest compute efficiency before starting the actual training process.

Model vs Data Parallelism Trade-off

There are two kinds of parallelism to exploit in large model training: data parallelism and model parallelism. In [301], I demonstrate that current ML vendors seem to build their hardware to efficiently optimize one specific type of parallelism. Both NVIDIA and Google rely on data parallelism to scale performance, whereas some startups, like Cerebras and Graphcore, take a different approach, focusing heavily on model parallelism. It seems that one open question researchers can help answer is what the right balance between model and data parallelism should be. Data parallelism is difficult to scale, and excessive hyperparameter tuning gives a clear advantage to more prominent players with the time and resources to tune them. In contrast, model parallelism requires less tuning, but as soon as the model no longer fits on-chip or on-node, most of the advantages are lost. Vendors seem to be coalescing around a hybrid solution, but where will the sweet spot be in the machine learning solutions of the future?

6.2.2 Phase-Aware Thread Scheduling for Micro-Second-Scale Latency

A key observation from modern data centers is the profound of micro-second latency resources [33] that cannot be hidden by the coarse-granularity of OS context switching, e.g., NVM memory or dis-aggregated DRAM. One solution to improve the latency hiding of data-intensive services for micro-second scale resources is via fast hardware-support context switching for many running threads (or requests), similar to GPU's warp scheduling to hide memory latency. However, running and interleaving many threads on the same CPU core can cause severe cache contention, exaggerating service latency and decreasing energy efficiency. In this project, I aim to study the execution phases of microservice workloads and quantify the cache footprint over time. Using this information, I plan to co-design the RPU's hardware and software to build efficient thread scheduling techniques by interleaving the high-cache footprint compute phase of the running thread with smaller footprint phases from other threads in the pool, and hence hiding micro-second latency and alleviating cache contention.

6.2.3 Physical Design of RPU with RISC-V Prototyping

In Chapter 4, I propose the Request Processing Unit (RPU), which modifies out-of-order CPU cores to execute microservices using a SIMT execution model. Through performance modeling simulation, the RPU system achieves 5.6x higher throughput/Watt while maintaining acceptable service latency for contemporary microservices. However, for more accurate evaluation and to prove the actual benefits of the RPU to the research community and data center providers, I am motivated to build and physically layout the RPU chip and test it in a real environment. The project will rely on the open-source RISC-V ISA for hardware implementation along with Apache HTTP server and LLVM compiler for software stack. Through the prototype, this work will evaluate the fundamental principles of RPU, tackle implementation related issues that were not addressed in the simulation experiment (e.g. branch predication penalty and TLB duplication overhead for large-footprint services), and study the limitations of RPU-based systems.

REFERENCES

[1] *Top500*, https://www.top500.org/lists/top500/.

[2] M. Khairy, A. G. Wassal, and M. Zahran, "A survey of architectural approaches for improving GPGPU performance, programmability and heterogeneity," *Journal of Parallel and Distributed Computing*, 2019.

[3] A. Arunkumar, E. Bolotin, B. Cho, *et al.*, "MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 320–332.

[4] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and Programmability," *IEEE Micro 2018*, vol. 38, no. 2, pp. 42–52,

[5] T. Vijayaraghavany, Y. Eckert, G. H. Loh, *et al.*, "Design and Analysis of an APU for Exascale Computing," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 85–96.

[6] O. Villa, D. R. Johnson, M. Oconnor, *et al.*, "Scaling the Power Wall: A Path to Exascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014, pp. 830–841.

[7] A. Kannan, N. E. Jerger, and G. H. Loh, "Exploiting Interposer Technologies to Disintegrate and Reintegrate Multicore Processors," *IEEE Micro*, pp. 84–93, 2016.

[8] P. Bright, *Moore's law really is dead this time*, https://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time/, 2016.

[9] T. Simonite, *Moore's Law Is Dead. Now What?* https://www.technologyreview.com/s/601441/moores-law-is-dead-now-what/, 2016.

[10] U. Milic, O. Villa, E. Bolotin, *et al.*, "Beyond the Socket: NUMA-aware GPUs," in *Proceedings of the 50th Annual International Symposium on Microarchitecture (MICRO)*, 2017, pp. 123–135.

[11] L. A. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines, 3rd edition," *Synthesis lectures on computer architecture*, 2018.

[12] J. Haj-Yihia, A. Yasin, Y. B. Asher, and A. Mendelson, "Fine-grain power breakdown of modern out-of-order cores and its implications on skylake-based systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[13] M. D. Powell, A. Biswas, J. S. Emer, S. S. Mukherjee, B. R. Sheikh, and S. Yardi, "Camp: A technique to estimate per-structure power at run-time using a few simple parameters," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, IEEE, 2009, pp. 289–300.

[14] Y. Sverdlik, *Growth of Hyperscale Data Centers*, https://www.datacenterknowledge.com/cloud/analysts-there-are-now-more-500-hyperscale-data-centers-world, 2019.

[15] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, 2019.

[16] C. Trueman, *Why data centres are the new frontier in the fight against climate change*, https://www.computerworld.com/article/3431148/why-data-centres-are-the-new-frontier-in-the-fight-against-climate-change.html, 2019.

[17] D. Patterson, J. Gonzalez, Q. Le, *et al.*, "Carbon emissions and large neural network training," *arXiv preprint arXiv:2104.10350*, 2021.

[18] I. King, A. Leung, and D. Pogkas, *The Chip Shortage Keeps Getting Worse. Why Can't We Just Make More?* https://www.bloomberg.com/graphics/2021-chip-production-why-hard-to-make-semiconductors/, 2021.

[19] N. Jones, *How to stop data centres from gobbling up the world's electricity*, https://www.nature.com/articles/d41586-018-06610-y.

[20] N. P. Jouppi, D. H. Yoon, G. Kurian, *et al.*, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, 2020.

[21] A. M. Caulfield, E. S. Chung, A. Putnam, *et al.*, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[22] N. P. Jouppi, D. H. Yoon, M. Ashcraft, *et al.*, "Ten lessons from three generations shaped google's tpuv4i: Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[23] W. J. Dally, J. Balfour, D. Black-Shaffer, *et al.*, "Efficient embedded computing," *Computer*, 2008.

[24] J. Sartori, B. Ahrens, and R. Kumar, "Power balanced pipelines," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012.

[25] Z. Xie, X. Xu, M. Walker, *et al.*, "APOLLO: An Automated Power Modeling Framework for Runtime Power Introspection in High-Volume Commercial Microprocessors," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[26] A. Sriraman and T. F. Wenisch, "μ Suite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[27] Y. Gan, Y. Zhang, D. Cheng, *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[28] S. Ibanez, A. Mallery, S. Arslan, *et al.*, "The nanoPU: Redesigning the CPU-Network Interface to Minimize RPC Tail Latency," *arXiv preprint arXiv:2010.12114*, 2020.

[29] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: efficient and fast RPCs in cloud microservices with near-memory reconfigurable NICs," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[30] A. Sriraman, A. Dhanotia, and T. F. Wenisch, "SoftSKU: Optimizing server architectures for microservice diversity@ scale," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[31] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[32] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for OLDI microservices," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018.

[33] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, 2017.

[34] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing server efficiency in the face of killer microseconds," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[35] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the killer microsecond," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[36] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 372–383.

[37] NVIDIA, *CuDNN Devloper Guide*, https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html, 2017.

[38] D. Wong, N. S. Kim, and M. Annavaram, "Approximating warps with intra-warp operand value similarity," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 176–187.

[39] M. D. Sinclair, J. Alsop, and S. V. Adve, "Chasing Away RAts: Semantics and evaluation for relaxed atomics on heterogeneous systems," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 161–174.

[40] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Performance Analysis of Systems and Software, 2009. IEEE International Symposium on*, IEEE, 2009, pp. 163–174.

[41] NVIDIA, *Volta V100 White paper*, http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[42] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, IEEE, 2010, pp. 235–246.

[43] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal, "A detailed GPU cache model based on reuse distance theory," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, IEEE, 2014, pp. 37–48.

[44] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.

[45] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[46] T. M. Aamodt, W. W. Fung, I. Singh, *et al.*, *GPGPU-Sim 3. x manual*, 2016.

[47] A. Jain, M. Khairy, and T. G. Rogers, "A Quantitative Evaluation of Contemporary GPU Simulation Methodology," *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)*, vol. 2, no. 2, p. 35, 2018.

[48] B. Beckmann and A. Gutierrez, "The AMD GEM5 APU Simulator: Modeling Heterogeneous Systems in GEM5," in *Tutorial at the International Symposium on Microarchitecture (MICRO)*, 2015.

[49] A. Gutierrez, B. M. Beckmann, A. Dutu, *et al.*, "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 608–619.

[50] Y. Sun, T. Baruah, S. A. Mojumder, *et al.*, "MGPUSim: Enabling Multi-GPU Performance Modeling and Optimization," in *Proceedings of the 46th International Symposium on Computer Architecture (ISCA)*, 2019, pp. 197–209.

[51] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "MacSim: A CPU-GPU heterogeneous simulation framework user guide," *Technical Report, Georgia Institute of Technology*, 2012.

[52] P. Gera, H. Kim, H. Kim, S. Hong, V. George, and C.-K. C. Luk, "Performance characterisation and simulation of intel's integrated GPU architecture," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 139–148.

[53] X. Gong, R. Ubal, and D. Kaeli, "Multi2Sim Kepler: A detailed architectural GPU simulator," in *Performance Analysis of Systems and Software (ISPASS), 2017 IEEE International Symposium on*, IEEE, 2017, pp. 269–278.

[54] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: A simulation framework for cpu-gpu computing," in *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*, IEEE, 2012, pp. 335–344.

[55] J. Lew, D. A. Shah, S. Pati, *et al.*, "Analyzing Machine Learning Workloads Using a Detailed GPU Simulator," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 151–152.

[56] NVIDIA, *CUDA Binary Utilities*, https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html, 2017.

[57] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling Deep Learning Accelerator Enabled GPUs," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 79–92.

[58] Giroux, Olivier and Durant, Luke, *Inside Volta*, http://on-demand.gputechconf.com/gtc/2017/presentation/s7798-luke-durant-inside-volta.pdf, 2017.

[59] Nvidia, *CUDA Programming Guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[60] J. Liptay, "Structural aspects of the system/360 model 85," *Readings in computer architecture*, p. 373, 2000.

[61] A. Seznec, "Decoupled sectored caches: Conciliating low tag implementation cost," in *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, 1994.

[62] Paulius Micikevicius, *Kepler Architecture*, http://on-demand.gputechconf.com/gtc/2013/presentations/S3466-Programming-Guidelines-GPU-Architecture.pdf, 2013.

[63] A. M. Aji, M. Daga, and W.-c. Feng, "Bounding the effect of partition camping in GPU kernels," in *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF)*, 2011, p. 27.

[64] Y. Liu, X. Zhao, M. Jahre, *et al.*, "Get out of the valley: power-efficient address mapping for GPUs," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 166–179.

[65] B. R. Rau, "Pseudo-Randomly Interleaved Memory," in *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA)*, 1991, pp. 74–83.

[66] J. Standard, "High Bandwidth Memory (HBM) DRAM," *JESD235*, 2013.

[67] M. O'Connor, "Highlights of the High-Bandwidth Memory (HBM) standard," in *Memory Forum Workshop*, 2014.

[68] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John, "The virtual write queue: Coordinating dram and last-level cache policies," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 38, 2010, pp. 72–82.

[69] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems," *Technical Report, The University of Texas at Austin, TR-HPS-2010-002*, 2010.

[70] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing DRAM latency divergence in irregular GPGPU applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 128–139.

[71] R. H. Saavedra-Barrera, "CPU performance evaluation and execution time prediction using narrow spectrum benchmarking," *Technical Report No. UCB/CSD-92-684, University of California, Berkeley,*, 1993.

[72] S. Che, M. Boyer, J. Meng, *et al.*, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[73] NVIDIA, *CUDA C/C++ SDK Code Samples*, http://developer.nvidia.com/cuda-cc-sdk-code-samples.

[74] J. A. Stratton, C. Rodrigues, I.-J. Sung, *et al.*, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[75] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a High-Level Language Targeted to GPU Codes," in *Innovative Parallel Computing (InPar)*, 2012, pp. 1–10.

[76] NVIDIA, *CUTLASS: CUDA template library for dense linear algebra at all levels and scales*, https://github.com/NVIDIA/cutlass, 2018.

[77] S. Narang and G. Diamos, *Baidu Deepbench*, https://svail.github.io/DeepBench, 2016.

[78] NVIDIA, *Nvidia Profiler 8.0*, http://docs.nvidia.com/cuda/profiler-users-guide/index.html, 2017.

[79] NVIDIA, *NVIDIA Nsight CLI*, https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html, 2019.

[80] NVIDIA, *GeForce GTX Titan*, https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan/specifications, 2013.

[81] NVIDIA, *Pascal Titan X*, https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/, 2017.

[82] NVIDIA, *GeForce RTX 2060*, https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2060/, 2019.

[83] N. P. Jouppi, "Cache write policies and performance," in *20th Annual International Symposium on Computer Architecture (ISCA)*, 1993, pp. 191–201.

[84] M. Rhu, M. Sullivan, J. Leng, and M. Erez, "A locality-aware memory hierarchy for energy-efficient GPU architectures," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 86–98.

[85] K. Pagiamtzis and A. Sheikholeslami, “A low-power content-addressable memory (CAM) using pipelined hierarchical search scheme,” *IEEE Journal of Solid-State Circuits*, vol. 39, no. 9, pp. 1512–1519, 2004.

[86] J. Tuck, L. Ceze, and J. Torrellas, “Scalable cache miss handling for high memory-level parallelism,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2006, pp. 409–422.

[87] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Cache-Conscious Wavefront Scheduling,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[88] H. Zhu, M. Akrout, B. Zheng, *et al.*, “Benchmarking and Analyzing Deep Neural Network Training,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 88–100.

[89] W. Jia, K. A. Shaw, and M. Martonosi, “MRPB: Memory request prioritization for massively parallel processors,” in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, 2014.

[90] A. Sethia, D. A. Jamshidi, and S. Mahlke, “Mascar: Speeding up GPU warps by reducing memory pitstops,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[91] G. Koo, Y. Oh, W. W. Ro, and M. Annavaram, “Access pattern-aware cache management for improving data utilization in GPU,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[92] M. Khairy, M. Zahran, and A. Wassal, “SACAT: Streaming-Aware Conflict-Avoiding Thrashing-Resistant GPGPU Cache Management Scheme,” *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[93] M. Khairy, M. Zahran, and A. G. Wassal, “Efficient Utilization of GPGPU Cache Hierarchy,” in *Proceedings of the 8th Workshop on General Purpose Processing using GPUS*, 2015.

[94] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, “Memory access scheduling,” in *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*, 2000, pp. 128–138.

[95] JEDEC Standard, “GDDR5X,” *JESD232A*, 2016.

[96] S. Pal, D. Petrisko, M. Tomei, P. Gupta, S. S. Iyer, and R. Kumar, “Architecting Waferscale Processors-A GPU Case Study,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 250–263.

[97] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, "AMD Chiplet Architecture for High-Performance Server and Desktop Products," in *IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 44–45.

[98] N. E. Jerger, A. Kannan, Z. Li, and G. H. Loh, "NoC Architectures for Silicon Interposer Systems: Why Pay for more Wires when you Can Get them (from your interposer) for Free?" In *Proceedings of the 47th Annual International Symposium on Microarchitecture (MICRO)*, 2014, pp. 458–470.

[99] A. Kannan, N. E. Jerger, and G. H. Loh, "Enabling Interposer-based Disintegration of Multi-Core Processors," in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 546–558.

[100] G. H. Loh, N. E. Jerger, A. Kannan, and Y. Eckert, "Interconnect-Memory Challenges for Multi-chip, Silicon Interposer Systems," in *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS)*, 2015, pp. 3–10.

[101] A. Arunkumar, E. Bolotin, D. Nellans, and C.-J. Wu, "Understanding the Future of Energy Efficiency in Multi-Module GPUs," in *IEEE 25th International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 519–532.

[102] H. Kim, R. Hadidi, L. Nai, *et al.*, "CODA: Enabling Co-location of Computation and Data for Multiple GPU Systems," *ACM Transactions on Architecture and Code Optimization (TACO)*, pp. 1–23, 2018.

[103] W. Li, "Compiling for NUMA Parallel Machines," Cornell University, Tech. Rep., 1994.

[104] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann San Francisco, 2002, vol. 289.

[105] J. W. Poulton, W. J. Dally, X. Chen, *et al.*, "A 0.54 pJ/b 20 Gb/s Ground-Referenced Single-Ended Short-Reach Serial Link in 28 nm CMOS for Advanced Packaging Applications," *IEEE Journal of Solid-State Circuits*, pp. 3206–3218, 2013.

[106] Y. S. Shao, J. Clemons, R. Venkatesan, *et al.*, "Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 14–27.

[107] Intel, *Intel EMIB*, https://www.intel.com/content/www/us/en/foundry/emib.html/, 2016.

[108] S. Pal, D. Petrisko, A. A. Bajwa, P. Gupta, S. S. Iyer, and R. Kumar, "A Case for Packageless Processors," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 466–479.

[109] B. Solca, *NVIDIA DGX-2 is the world largest GPU*, https://www.notebookcheck.net/Nvidia-DGX-2-is-the-world-s-largest-GPU.292930.0.html, 2018.

[110] NVIDIA, *NVIDIA NVLink: High Speed GPU Interconnect*, https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/.

[111] B. Falsafi and D. A. Wood, "Reactive NUMA: A Design for Unifying S-COMA and CC-MAMA," in *Proceedings of the 24th annual international symposium on Computer architecture (ISCA)*, 1997, pp. 229–240.

[112] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple But Effective Techniques for NUMA Memory Management," *ACM SIGOPS Operating Systems Review*, pp. 19–31, 1989.

[113] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin, "An Argument for Simple COMA," in *First IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 1995, pp. 276–285.

[114] M. Dashti, A. Fedorova, J. Funston, *et al.*, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 381–394.

[115] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches," in *Proceedings of the 36th annual international symposium on Computer architecture (ISCA)*, 2009, pp. 184–195.

[116] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi, "Application-to-Core Mapping Policies to Reduce Memory Interference in Multi-Core Systems," in *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 107–118.

[117] D. Tam, R. Azimi, and M. Stumm, "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors," in *ACM SIGOPS Operating Systems Review*, 2007, pp. 47–58.

[118] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A Case for NUMA-aware Contention Management on Multicore Systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, 2010, pp. 557–558.

[119] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and Loop Scheduling on NUMA Multiprocessors," in *International Conference on Parallel Processing (ICPP)*, 1993, pp. 140–147.

[120] V. Young, A. Jaleel, E. Bolotin, E. Ebrahimi, D. Nellans, and O. Villa, "Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems," in *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 339–351.

[121] T. Zheng, D. Nellans, A. Zulfiqar, M. Stephenson, and S. W. Keckler, "Towards High Performance Paged Memory for GPUs," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 345–357.

[122] N. Agarwal, D. Nellans, M. O'Connor, S. W. Keckler, and T. F. Wenisch, "Unlocking Bandwidth for GPUs in CC-NUMA Systems," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 354–365.

[123] T. Baruah, Y. Sun, A. T. Dinçer, *et al.*, "Griffin: Hardware-Software Support for Efficient Page Migration in Multi-GPU Systems," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 596–609.

[124] N. Vijaykumar, E. Ebrahimi, K. Hsieh, P. B. Gibbons, and O. Mutlu, "The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality In GPUs," in *45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 829–842.

[125] A. Li, S. L. Song, W. Liu, X. Liu, A. Kumar, and H. Corporaal, "Locality-Aware CTA Clustering for Modern GPUs," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 297–311.

[126] J. Cabezas, L. Vilanova, I. Gelado, T. B. Jablin, N. Navarro, and W.-m. W. Hwu, "Automatic Parallelization of Kernels in Shared-Memory Multi-GPU Nodes," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 3–13.

[127] NVIDIA, *NVIDIA NVSWITCH*, https://images.nvidia.com/content/pdf/nvswitch-technical-overview.pdf.

[128] D. Kirk and W. Wen-mei, *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 2010.

[129] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 157–166.

[130] A. Jog, O. Kayiran, A. K. Mishra, *et al.*, "Orchestrated Scheduling and Prefetching for GPGPUs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013, pp. 332–343.

[131] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, *et al.*, “OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU Performance,” in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013, pp. 395–406.

[132] T. G. Rogers, M. O’Connor, and T. M. Aamodt, “Divergence-aware Warp Scheduling,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 99–110.

[133] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling,” in *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[134] X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, “HMG: Extending Cache Coherence Protocols Across Modern Hierarchical Multi-GPU Systems,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 582–595.

[135] M. A. O’Neil and M. Burtscher, “Microarchitectural Performance Characterization of Irregular GPU Kernels,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 130–139.

[136] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, “Pannotia: Understanding Irregular GPGPU Graph Applications,” in *IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 185–195.

[137] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, “Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism,” *arXiv preprint arXiv:1909.08053*, 2019.

[138] S. Han, H. Mao, and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding,” *arXiv preprint arXiv:1510.00149*, 2015.

[139] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, “Efficient processing of Deep Neural Networks,” *Synthesis Lectures on Computer Architecture*, 2020.

[140] J. Appleyard, T. Kocisky, and P. Blunsom, “Optimizing Performance of Recurrent Neural Networks on GPUs,” *arXiv preprint arXiv:1604.01946*, 2016.

[141] NVIDIA, *cuBLASXt*, https://docs.nvidia.com/cuda/cublas/index.html/using-the-cublasXt-api, 2020.

[142] S. Kanev, J. P. Darago, K. Hazelwood, *et al.*, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[143] M. Ferdman, A. Adileh, O. Kocberber, *et al.*, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[144] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory Hierarchy for Web Search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[145] G. Ayers, N. P. Nagendra, D. I. August, *et al.*, "AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[146] G. Ayers, H. Litz, C. Kozyrakis, and P. Ranganathan, "Classifying Memory Access Patterns for Prefetching," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[147] D. Gope, D. J. Schlais, and M. H. Lipasti, "Architectural support for server-side PHP processing," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[148] P. Lotfi-Kamran, B. Grot, M. Ferdman, *et al.*, "Scale-Out Processors," *2012 ACM/IEEE 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[149] K. Lepak, G. Talbot, S. White, N. Beck, S. Naffziger, S. FELLOW, *et al.*, "The next generation AMD enterprise server product architecture," *IEEE hot chips*, 2017.

[150] M. Arafa, B. Fahim, S. Kottapalli, *et al.*, "Cascade Lake: Next generation Intel XEON scalable processor," *IEEE Micro*, 2019.

[151] A. Pellegrini, N. Stephens, M. Bruce, *et al.*, "The ARM Neoverse N1 platform: Building blocks for the next-gen cloud-to-edge infrastructure SoC," *IEEE Micro*, 2020.

[152] *Ampere Computing*, https://amperecomputing.com/altra/.

[153] R. Sugumar, M. Shah, and R. Ramirez, "Marvell ThunderX3: Next-Generation Arm-Based Server Processor," *IEEE Micro*, 2021.

[154] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke, "IBM Power9 processor architecture," *IEEE Micro*, 2017.

[155] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "Memcachedgpu: Scaling-up scale-out key-value stores," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[156] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck, "Rhythm: Harnessing Data Parallel Hardware for Server Workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[157] M. Silberstein, B. Ford, I. Keidar, and E. Witchel, "GPUfs: Integrating a file system with GPUs," in *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[158] M. Silberstein, S. Kim, S. Huh, *et al.*, "GPUnet: Networking abstractions for GPU programs," *ACM Transactions on Computer Systems (TOCS)*, 2016.

[159] J. Veselỳ, A. Basu, A. Bhattacharjee, G. H. Loh, M. Oskin, and S. K. Reinhardt, "Generic system calls for GPUs," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[160] NVIDIA, *NVIDIA GPUDirect*, https://developer.nvidia.com/gpudirect.

[161] A. Fog *et al.*, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs," *Copenhagen University College of Engineering*, vol. 93, p. 110, 2011.

[162] Amazon, *Netflix on AWS*, https://aws.amazon.com/solutions/case-studies/netflix/, 2021.

[163] V. Agrawal, M. A. Dinani, Y. Shui, M. Ferdman, and N. Honarmand, "Massively Parallel Server Processors," *IEEE Computer Architecture Letters*, 2019.

[164] Memchached, *Caching beyond RAM: Riding the cliff*, https://memcached.org/blog/nvm-multidisk/, 2019.

[165] A. Basak, Z. Qu, J. Lin, *et al.*, "Improving streaming graph processing performance using input knowledge," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[166] D. Meisner and T. F. Wenisch, "DreamWeaver: Architectural Support for Deep Sleep," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[167] D. Meisner, B. T. Gold, and T. F. Wenisch, "PowerNap: Eliminating Server Idle Power," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[168] C.-H. Chou, L. N. Bhuyan, and D. Wong, "μDPM: Dynamic power management for the microsecond era," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[169] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong, "Multi-execution: Multicore caching for data-similar executions," in *2009 ACM/IEEE 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[170] U. Hölzle, "Brawny cores still beat wimpy cores, most of the time," *IEEE MICRO*, 2010.

[171] V. Petrucci, M. A. Laurenzano, J. Doherty, *et al.*, "Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[172] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Communications of the ACM*, 2018.

[173] J. Hruska, *DDR5 specifications*, https://www.extremetech.com/computing/312730-ddr5-memory-specification-finalized-up-to-6400gt-s-2tb-lrdimms.

[174] S. Schlachter and B. Drake, *Introducing Micron DDR5 SDRAM: More Than a Generational Update*, 2020.

[175] R. Press, *DDR5 vs DDR4 – All the Design Challenges & Advantages*, https://www.rambus.com/blogs/get-ready-for-ddr5-dimm-chipsets/, 2021.

[176] T. Morgan, *NVIDIA enters the ARMs race with homegrown Grace CPUS*, https://www.nextplatform.com/2021/04/12/nvidia-enters-the-arms-race-with-homegrown-grace-cpus/, 2021.

[177] Z. Peterson, *DDR5 vs. DDR6: Here's What to Expect in RAM Modules*, https://resources.altium.com/p/ddr5-vs-ddr6-heres-what-expect-ram-modules, 2021.

[178] I. Cutress, *Intel to Launch Next-Gen Sapphire Rapids Xeon with High Bandwidth Memory*, https://www.anandtech.com/show/16795/intel-to-launch-next-gen-sapphire-rapids-xeon-with-high-bandwidth-memory, 2021.

[179] A. Tino, C. Collange, and A. Seznec, "SIMT-X: Extending single-instruction multi-threading to out-of-order cores," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2020.

[180] T. M. Aamodt, W. W. L. Fung, and T. G. Rogers, “General-Purpose Graphics Processor Architectures,” *Synthesis Lectures on Computer Architecture*, 2018.

[181] B. W. Coon, J. E. Lindholm, P. C. Mills, and J. R. Nickolls, *Processing an indirect branch instruction in a simd architecture*, US Patent 7,761,697, Jul. 2010.

[182] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, “Dynamic warp formation and scheduling for efficient GPU control flow,” in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[183] A. ElTantawy and T. M. Aamodt, “MIMD synchronization on SIMT architectures,” in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016, pp. 1–14.

[184] B. Kerbl, M. Kenzel, M. Winter, and M. Steinberger, “CUDA and Applications to Task-based Programming,” in *Eurographics (Tutorials)*, 2022.

[185] NVIDIA, “NVIDIA Tesla V100 GPU architecture,” 2017.

[186] L. Nyland, J. R. Nickolls, G. Hirota, and T. Mandal, *Systems and methods for coalescing memory accesses of parallel threads*, US Patent 8,086,806, Dec. 2011.

[187] J. Leng, T. Hetherington, A. ElTantawy, *et al.*, “GPUWattch: enabling energy optimizations in GPGPUs,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[188] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi, “Dynamic speculation and synchronization of data dependences,” in *Proceedings of the 24th annual international symposium on Computer architecture*, 1997.

[189] felixcloutier.com, *INVLPG — Invalidate Specific TLB Entries*, https://www.felixcloutier.com/x86/invlpg.

[190] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, “Improving GPU performance via large warps and two-level warp scheduling,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[191] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanović, “Convergence and scalarization for data-parallel architectures,” in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[192] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Communications of the ACM*, 2013.

[193] B. A. Hechtman, S. Che, D. R. Hower, *et al.*, "QuickRelease: A throughput-oriented approach to release consistency on GPUs," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[194] ARM, *Overview of memory consistency*, https://developer.arm.com/documentation/ddi0406/c/Appendices/Barrier-Litmus-Tests/Introduction/Overview-of-memory-consistency.

[195] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell, "Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8," *Proceedings of the ACM on Programming Languages*, 2017.

[196] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding power multiprocessors," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 175–186.

[197] L. Maranget, S. Sarkar, and P. Sewell, "A tutorial introduction to the ARM and POWER relaxed memory models," *Technical Report*, 2012.

[198] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE journal of solid-state circuits*, 2006.

[199] M. Clark, "A new x86 core architecture for the next generation of computing," in *2016 IEEE Hot Chips 28 Symposium (HCS)*, 2016.

[200] J. Meng, J. W. Sheaffer, and K. Skadron, "Exploiting inter-thread temporal locality for chip multithreading," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, IEEE, 2010, pp. 1–12.

[201] H. Kwak, B. Lee, A. R. Hurson, S.-H. Yoon, and W.-J. Hahn, "Effects of multithreading on cache performance," *IEEE Transactions on Computers*, 1999.

[202] R. T. Fielding and G. Kaiser, "The Apache HTTP server project," *IEEE Internet Computing*, 1997.

[203] A. Wiggins and J. Langston, "Enhancing the scalability of MemCached," *Intel document, unpublished*, 2012.

[204] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications," *ACM Sigplan Notices*, 2000.

[205] I. Gelado and M. Garland, "Throughput-oriented GPU memory allocation," in *Proceedings of the 24th symposium on principles and practice of parallel programming*, 2019.

[206] P. Menage, *TCMalloc : Thread-Caching Malloc*, http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[207] S. Damani, D. R. Johnson, M. Stephenson, *et al.*, "Speculative reconvergence for improved SIMT efficiency," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, 2020.

[208] NVIDIA, "CUDA C Programming Guide,"

[209] C. Wellons, *Raw Linux Threads via System Calls*, https://nullprogram.com/blog/2015/05/15/, 2015.

[210] M. Kerrisk, *mmap — Linux manual page*, https://man7.org/linux/man-pages/man2/mmap.2.html.

[211] M. Anderson, B. Chen, S. Chen, *et al.*, "First-generation inference accelerator deployment at facebook," *arXiv preprint arXiv:2107.04140*, 2021.

[212] M. Winter, M. Parger, D. Mlakar, and M. Steinberger, "Are dynamic memory managers on GPUs slow? a survey and benchmarks," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021, pp. 219–233.

[213] A. González, M. Valero, N. Topham, and J. M. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *Proceedings of the 11th international conference on Supercomputing*, 1997, pp. 76–83.

[214] M. Kerrisk, *ld.so — Linux manual page*, https://man7.org/linux/man-pages/man8/ld.so.8.html.

[215] StackOverflow, *What is the LDPRELOAD trick?* https://stackoverflow.com/questions/426230/what-is-the-ld-preload-trick.

[216] J. Meng, D. Tarjan, and K. Skadron, "Dynamic warp subdivision for integrated branch and memory divergence tolerance," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010.

[217] M. Rhu and M. Erez, "The Dual-Path Execution Model for Efficient GPU Control Flow," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[218] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A Scalable Multi-Path Microarchitecture for Efficient GPU Control Flow," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[219] A. Basak, J. Lin, R. Lorica, *et al.*, "SAGA-bench: Software and hardware characterization of streaming graph analytics workloads," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[220] Google, *gRPC: A high performance, open source universal RPC framework*, https://grpc.io/.

[221] C.-K. Luk, R. Cohn, R. Muth, *et al.*, "PIN: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[222] N. Jiang, D. U. Becker, G. Michelogiannakis, *et al.*, "A detailed and flexible cycle-accurate network-on-chip simulator," in *2013 IEEE international symposium on performance analysis of systems and software (ISPASS)*, IEEE, 2013, pp. 86–96.

[223] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture-ISCA*, Association for Computing Machinery, vol. 13, 2013, pp. 23–27.

[224] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," *HP laboratories*, 2009.

[225] Y. Zhang, Y. Gan, and C. Delimitrou, "μQsim: Enabling accurate and scalable simulation for interactive microservices," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[226] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, 2009.

[227] J. Zhai, C. Bai, B. Zhu, Y. Cai, Q. Zhou, and B. Yu, "McPAT-Calib: A Microarchitecture Power Modeling Framework for Modern CPUs," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021.

[228] Nvidia, *A100 Tensor Core GPU architecture*, 2020.

[229] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, 2013.

[230] D. Kusswurm, *Modern X86 Assembly Language Programming.* Springer, 2014.

[231] *x86/x64 SIMD Instruction List (SSE to AVX512)*, https://www.officedaytime.com/simd512e/.

[232] N. Stephens, S. Biles, M. Boettcher, *et al.*, "The ARM scalable vector extension," *IEEE micro*, 2017.

[233] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in *2012 Innovative Parallel Computing (InPar)*, 2012.

[234] *Which are the most used web servers?* https://www.stackscale.com/blog/top-web-servers/.

[235] *What is the python global interpreter lock (gil)*, https://realpython.com/python-gil/.

[236] M. P. Puig, L. De Giusti, and M. Naiouf, "Are GPUs Non-Green Computing Devices?" *Journal of Computer Science and Technology*, 2018.

[237] V. W. Lee, C. Kim, J. Chhugani, *et al.*, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th annual international symposium on Computer architecture*, 2010, pp. 451–460.

[238] Xcelerit, *Computing benchmarks: Processors*, https://www.xcelerit.com/computing-benchmarks/processors/.

[239] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," in *2019 IEEE international conference on embedded software and systems (ICESS)*, 2019.

[240] "CUDA C Programming Guide." (2017), [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[241] G. Thomas-Collignon and V. Mehta, *Optimizing CUDA Applications for NVIDIA A100 GPU*, https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21819-optimizing-applications-for-nvidia-ampere-gpu-architecture.pdf.

[242] NVIDIA, *Kepler GPU White Paper*, https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[243] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A Heterogeneous CPU-GPU Simulator," *Computer Architecture Letters*, 2014.

[244] V. M. Del Barrio, C. González, J. Roca, A. Fernández, and E. Espasa, "ATTILA: a cycle-level execution-driven simulator for modern GPU architectures," in *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*, IEEE, 2006, pp. 231–241.

[245] A. A. Gubran and T. M. Aamodt, "Emerald: Graphics Modeling for SoC Systems," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 169–182.

[246] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–12.

[247] M. Stephenson, S. K. Sastry Hari, Y. Lee, *et al.*, "Flexible software profiling of GPU architectures," in *ACM SIGARCH Computer Architecture News*, ACM, vol. 43, 2015, pp. 185–197.

[248] X. Wang, K. Huang, A. Knoll, and X. Qian, "A Hybrid Framework for Fast and Accurate GPU Performance Estimation through Source-Level Analysis and Trace-Based Simulation," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 506–518.

[249] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, IEEE, 2015, pp. 564–576.

[250] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, pp. 725–737.

[251] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "GPUMech: GPU performance modeling technique based on interval analysis," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 268–279.

[252] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 382–393.

[253] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proceedings of the 37th annual international symposium on computer architecture (ISCA)*, 2010, pp. 280–289.

[254] T. Nowatzki, J. Menon, C.-H. Ho, and K. Sankaralingam, "gem5, GPGPUsim, McPAT, GPUWattch," your favorite simulator here" considered harmful," 2014.

[255] H. Dai, C. Li, Z. Lin, and H. Zhou, "The Demand for a Sound Baseline in GPU Memory Architecture Research," 2017.

[256] A. Adileh, C. González-Alvarez, J. M. D. H. Ruiz, and L. Eeckhout, "Racing to hardware-validated simulation," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2019, pp. 58–67.

[257] R. Desikan, D. Burger, and S. W. Keckler, "Measuring Experimental Error in Microprocessor Simulation," in *Proceedings of the 28th annual International Symposium on Computer Architecture (ISCA)*, 2001, pp. 266–277.

[258] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, *et al.*, "Sources of Error in Full-System Simulation," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 13–22.

[259] A. M. Aji, L. S. Panwar, F. Ji, *et al.*, "On the Efficacy of GPU-Integrated MPI for Scientific Applications," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, 2013, pp. 191–202.

[260] V. V. Kindratenko, J. J. Enos, G. Shi, *et al.*, "GPU Clusters for High-Performance Computing," in *2009 IEEE International Conference on Cluster Computing and Workshops*, 2009, pp. 1–8.

[261] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a Single Compute Device Image in OpenCL for Multiple GPUs," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming (PPOPP)*, 2011, pp. 277–288.

[262] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent CPU-GPU Collaboration for Data-Parallel Kernels on Heterogeneous Systems," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques (PACT)*, 2013, pp. 245–256.

[263] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang, "Coordinated Static and Dynamic Cache Bypassing for GPUs," in *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 76–88.

[264] M. Lee, S. Song, J. Moon, *et al.*, "Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, 2014, pp. 260–271.

[265] L.-J. Chen, H.-Y. Cheng, P.-H. Wang, and C.-L. Yang, "Improving GPGPU Performance via Cache Locality Aware Thread Block Scheduling," *IEEE Computer Architecture Letters*, pp. 127–131, 2017.

[266] C. Li, R. Ausavarungnirun, C. J. Rossbach, *et al.*, "A Framework for Memory Oversubscription Management in Graphics Processing Units," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 49–63.

[267] H. Kim, J. Sim, P. Gera, R. Hadidi, and H. Kim, "Batch-Aware Unified Memory Management in GPUs for Irregular Workloads," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1357–1370.

[268] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU Compiler for Memory Optimization and Parallelism Management," in *InProceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 86–97.

[269] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs," in *Proceedings of the 22nd annual international conference on Supercomputing*, 2008, pp. 225–234.

[270] J. Shirako, A. Hayashi, and V. Sarkar, "Optimized Two-Level Parallelization for GPU Accelerators using the Polyhedral Model," in *Proceedings of the 26th International Conference on Compiler Construction*, 2017, pp. 22–33.

[271] Y. Paek and D. A. Padua, "Experimental Study of Compiler Techniques for NUMA Machines," in *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, 1998, pp. 187–193.

[272] J. Juega, J. Gomez, C. Tenllado, S. Verdoolaege, A. Cohen, and F. Catthoor, "Evaluation of state-of-the-art polyhedral tools for automatic code generation on GPUs," *XXIII Jornadas de Paralelismo, Univ. Complutense de Madrid*, 2012.

[273] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs," in *International Conference on Compiler Construction*, 2010, pp. 244–263.

[274] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly: Performing Polyhedral Optimizations on a Low-Level Intermediate Representation," *Parallel Processing Letters*, pp. 1–27, 2012.

[275] C. Bastoul, "Code Generation in the Polyhedral Model Is Easier Than You Think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004, pp. 7–16.

[276] NVIDIA, *NVCC*, https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[277] Z. Majo and T. R. Gross, "Matching Memory Access Patterns and Data Placement for NUMA Systems," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*, 2012, pp. 230–241.

[278] C. McCurdy and J. Vetter, "Memphis: Finding and Fixing NUMA-related Performance Problems on Multi-core Platforms," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010, pp. 87–96.

[279] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "ForestGOMP: an Efficient OpenMP Environment for NUMA Architectures," *International Journal of Parallel Programming*, pp. 418–439, 2010.

[280] M. Diener, E. H. Cruz, M. A. Alves, P. O. Navaux, and I. Koren, "Affinity-Based Thread and Data Mapping in Shared Memory Systems," *ACM Computing Surveys (CSUR)*, 2017.

[281] Y. Li, R. Melhem, A. Abousamra, and A. K. Jones, "Compiler-assisted Data Distribution for Chip Multiprocessors," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 501–512.

[282] G. Piccoli, H. N. Santos, R. E. Rodrigues, C. Pousa, E. Borin, and F. M. Quintão Pereira, "Compiler Support for Selective Page Migration in NUMA Architectures," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014, pp. 369–380.

[283] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large Pages May Be Harmful on NUMA Systems," in *Proceedings of 2014 USENIX Annual Technical Conference (USENIX ATC)*, 2014, pp. 231–242.

[284] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *The International Journal of High Performance Computing Applications*, pp. 110–124, 2012.

[285] A. Muddukrishna, P. A. Jonsson, and M. Brorsson, "Locality-Aware Task Scheduling and Data Distribution for OpenMP Programs on NUMA Systems and Manycore Processors," *Scientific Programming*, 2015.

[286] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "Characterizing and evaluating a key-value store application on heterogeneous CPU-GPU systems," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, 2012.

[287] S. Kalathingal, S. Collange, B. N. Swamy, and A. Seznec, "DITVA: Dynamic interthread vectorization architecture," *Journal of Parallel and Distributed Computing*, 2018.

[288] K. Kim, S. Lee, M. K. Yoon, G. Koo, W. W. Ro, and M. Annavaram, "Warped-preexecution: A GPU pre-execution approach for improving latency hiding," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, 2016, pp. 163–175.

[289] X. Gong, X. Gong, L. Yu, and D. Kaeli, "HAWS: Accelerating GPU wavefront execution through selective out-of-order execution," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 16, no. 2, pp. 1–22, 2019.

[290] K. Iliakis, S. Xydis, and D. Soudris, "LOOG: Improving GPU Efficiency With Light-Weight Out-Of-Order Execution," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 166–169, 2019.

[291] P. Xiang, Y. Yang, M. Mantor, N. Rubin, and H. Zhou, "Revisiting ILP designs for throughput-oriented GPGPU architecture," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE, 2015, pp. 121–130.

[292] R. Espasa, M. Valero, and J. E. Smith, "Out-of-Order Vector Architectures," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, IEEE, 1997, pp. 160–170.

[293] A. Pourhabibi, S. Gupta, H. Kassir, *et al.*, "Optimus Prime: Accelerating Data Transformation in Servers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1203–1216.

[294] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi, "Cerebros: Evading the RPC tax in datacenters," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 407–420.

[295] M. Khairy, J. Akshay, T. Aamodt, and T. G. Rogers, "Exploring Modern GPU Memory System Design Challenges through Accurate Modeling," *arXiv preprint arXiv:1810.07269*, 2018.

[296] C. Avalos Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, "Principal Kernel Analysis: A Tractable Methodology to Simulate Scaled GPU Workloads," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 724–737.

[297] V. Kandiah, S. Peverelle, M. Khairy, *et al.*, "AccelWattch: A Power Modeling Framework for Modern GPUs," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[298] M. Khairy, V. Nikiforov, D. Nellans, and T. G. Rogers, "Locality-Centric Data and Threadblock Management for Massive GPUs," in *2020 ACM/IEEE 53rd Annual International Symposium on Microarchitecture(MICRO)*, 2020.

[299] O. Villa, D. Lustig, Z. Yan, *et al.*, "Need for speed: Experiences building a trustworthy system-level gpu simulator," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, IEEE, 2021, pp. 868–880.

[300] M. Khairy, *Tpu vs gpu vs cerebras vs graphcore: A fair comparison between ml hardware*, https://medium.com/@khairy2011/tpu-vs-gpu-vs-cerebras-vs-graphcore-a-fair-comparison-between-ml-hardware-3f5a19d89e38, 2020.

[301] M. Khairy and T. Rogers, *An academic's attempt to clear the fog of the machine learning accelerator war*, https://www.sigarch.org/an-academics-attempt-to-clear-the-fog-of-the-machine-learning-accelerator-war/, 2020.